

Copyright © 1991, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**AUTOMATED DESIGN MANAGEMENT
USING TRACES**

by

Andrea Casotto

Memorandum No. UCB/ERL M91/22

15 March 1991

2000 1/1/91

**AUTOMATED DESIGN MANAGEMENT
USING TRACES**

by

Andrea Casotto

Memorandum No. UCB/ERL M91/22

15 March 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

**AUTOMATED DESIGN MANAGEMENT
USING TRACES**

by

Andrea Casotto

Memorandum No. UCB/ERL M91/22

15 March 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Automated Design Management Using Traces

Andrea Casotto

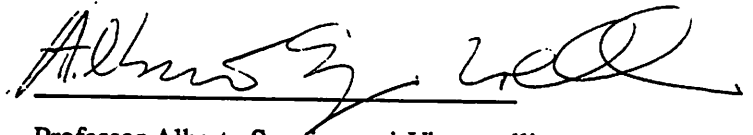
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley.

Abstract

The productivity of modern CAD systems can be increased with a layer of software, called the “automatic design manager,” whose goal is to provide services such as automatic sequencing and scheduling of the tools, coordination of team design and tracking of the design activity for documentation purposes.

An automatic management system for CAD is proposed, based on the idea that CAD tools can leave a “trace” of their execution. The trace is represented as a bipartite directed and acyclic graph in which the nodes represent either design data or CAD transactions. The trace is both a record of the design activity and a graph representing the dependencies among the design objects. The architecture of the system is distributed: a server manages the trace, while a number of clients can concurrently interact with the trace through the server. The system supports the notion of measurement on the design data, necessary to provide even more services such as tracking of design specifications, validation of design data, design estimation. The system is non-intrusive, because it does not affect the way designers interact with the tools.

The design manager has been implemented in a system called VOV. This prototype has been tested by many designers, including novices and experts. The results of these tests are reported.



Professor Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Acknowledgements

Finally a pause, a blank page to reflect upon the many people to whom I owe thanks.

My advisor, Alberto Sangiovanni-Vincentelli. He directed me towards design management and made available the resources for this research. Although I could never get enough of his time, I need to thank him for his guidance and his trust. I thank my other advisor, Prof. Richard Newton, for steering me away from the formalism of Petri nets and for his poignant criticism. He has contributed to form a large part of the ideas presented in this thesis. I consider myself lucky to work with these two stars of the CAD world. I only regret not having been able to defeat either of them in a tennis match.

Prof. Randy Katz is acknowledged for suggesting the use of the word “trace” in the context of design management and for helpful discussions in the early stages of this research. I thank Prof. John Wawrzynek for being the first to trust VOV and to use it in his classes. Prof. Alice Agogino has been kind accepting to be on my dissertation committee.

My office mates Mark Beardslee, Mitch Igusa and Chuck Kring, have been patient and helpful, they have tolerated my demos and even tried the earliest and buggiest versions of the system. Chuck has been especially cruel in dissecting a draft of this thesis, and I owe it to him if this thesis is now much more readable. The *Octtools* developers, in particular Rick Spiekelmier and David Harrison, have produced a tremendous amount of great code that now is also part of VOV. I thank them all. Gregg Whitcomb has invested a lot of energy into maintaining g++, which made the development of VOV so much simpler.

The students in CS292i, who were the first to use VOV in Spring 90, and the students in CS250, who were the first to use the assistant in Fall 90, have been instrumental in the progress of this research.

Flora Oviedo, Irena Stanczyk-Ng, and Elise Mills, who really run the show here in Cory Hall, have always been helpful in all office related matters. SRC has provided financial support throughout my five and a half years as Graduate student.

Contents

Table of Contents	iii
1 Introduction	1
1.1 A characterization of electronic design	3
1.2 Requirements of an automatic design manager	7
1.3 Background on the <i>Octtools</i>	10
1.3.1 The <i>Octtools</i>	10
1.3.2 Brief history of VOV and the <i>Octtools</i>	13
2 Previous Work	15
2.1 Representation of the design activity	16
2.1.1 Ad-hoc models for VLSI design	16
2.1.2 General models for design activity	20
2.1.3 Design environments	33
2.2 Intrusiveness of implementation	34
2.3 Classification of tools and data	35
2.4 Artificial intelligence techniques	37
2.5 Issues in data management	38
2.6 Commercial systems	40
2.7 Conclusion of the survey	41
3 The Design Trace	43
3.1 Design management based on design traces	43
3.1.1 Non-intrusive tracing	44
3.1.2 The trace	45
3.1.3 An example trace	48
3.2 The trace as a definitional language	50
3.2.1 Backtracking	51
3.3 Tools that run in place	52
3.4 The architecture	53
3.4.1 Communication between the tools and the server	55
3.4.2 Affinity of transitions	57
3.5 Interactive tools	59

3.6	The firing rule	60
3.7	Trace versus Petri net	61
3.8	Sets of nodes	62
3.8.1	Hierarchy in the trace	64
3.9	Services	67
3.9.1	Service: Design documentation	67
3.9.2	Service: data monitoring	70
3.9.3	Service: retracing	70
3.9.4	Service: Conflict detection	73
3.9.5	Management of refinements and alternatives	74
3.9.6	Archiving	76
3.10	Use of measurements	76
3.11	The assistant	78
3.12	Support of design methodology	84
3.13	Iteration in design	85
3.14	Principles that guided the development of VOV.	87
3.14.1	Simplicity	87
3.14.2	Non-intrusiveness	89
3.14.3	Distributed resources, localization of information	89
3.14.4	Focus on users	90
3.14.5	Emphasis on team design	91
3.14.6	No restriction to data visibility	92
3.14.7	Ignore design hierarchy	92
4	Implementation	96
4.1	The design trace	97
4.1.1	Attributes of nodes	97
4.1.2	Attributes of places and transitions	97
4.1.3	Canonical names for files	99
4.1.4	The representation of the trace	99
4.2	Special topics	102
4.2.1	Project identification	102
4.2.2	Robustness and safety	103
4.3	Software architecture	103
4.3.1	The hierarchy of classes	106
4.3.2	User interface	108
4.4	Performance	110
4.4.1	Server latency	110
4.4.2	Capsule overhead	111
4.4.3	Trace size	112
4.4.4	Small designs	113
4.4.5	Large designs	113

5	Experimental Results	115
5.1	Statistics on the design	115
5.2	BRIC	117
5.3	Floorplanning an FPU	118
5.4	Compilation of VOV	119
5.5	VOV in a VLSI design course	120
5.5.1	The laboratory exercises	120
5.5.2	The final project	126
5.5.3	Comment	127
6	Conclusion	129
A	Tutorial	132
A.1	Introduction	132
A.2	TUTORIAL: Design of a seven segment display driver	133
A.2.1	Start mini-VOV	133
A.2.2	Enter the assistant	134
A.2.3	The graphical interface	135
A.2.4	Getting assistance from the assistant	135
A.2.5	Your turn to act intelligent	138
A.2.6	The trace as a dependency graph	138
A.2.7	Validity of nodes	140
A.2.8	Automatic retracing	142
A.2.9	Review what has happened	142
A.2.10	Possible problems	144
A.2.11	Substitution a transition	145
A.2.12	Check the results	145
A.2.13	Try something new	145
A.2.14	Suspension or end of the exercise	148
A.3	TUTORIAL: Second part	148
A.3.1	What does vov_mini really do	148
A.3.2	Add many slaves to your server	149
A.3.3	Start the server	150
A.3.4	Clients	150
A.3.5	The event queue and the journal	151
A.3.6	The event queue	151
A.3.7	The trace	151
A.3.8	Annotations	152
A.3.9	Affinity of transition, interactive transitions	152
A.3.10	Graphical interface using vem/RPC	153
A.3.11	Status of the trace	153
A.3.12	Protection	154
A.3.13	Sets	154
A.3.14	Forgetting nodes	154
A.3.15	Moving stuff around the file system	154

A.3.16 Handy utilities	155
B Quick Tool Overview	156
Bibliography	157

Chapter 1

Introduction

The complexity of modern human artifacts such as microprocessors, aircraft, and satellites, demands the power of computer tools to assist the designers in many of the design tasks. This is particularly true in the electronic industry, which routinely deals with millions of components. In recent years, CAD systems have come a long way towards freeing the designers from the complexity of design, but designers now have to cope with the ever growing complexity of the CAD systems themselves.

Electronic design is the focus of this dissertation, because of the training of the author. However, an ill concealed ambition of this work is to be applicable in other domains of automatic design, that is wherever computers are used to assist the designers.

A “CAD system” is a collection of software programs that perform many tasks related to the analysis and synthesis of electronic systems, such as integrated digital circuits, microprocessor and printed circuit boards. Each autonomous component in a CAD system is a “tool.” CAD systems typically include, among others, tools to describe the behavior of electronic components, to simulate a circuit, to synthesize logic equations, to generate, optimize, and verify layout.

CAD systems take many forms, varying in user interface, capabilities, target technology, the number and complexity of the tools, and the way in which the tools communicate with each other. However, all systems seem to share this unwanted characteristic: they are difficult to learn, to use, and to master. There is a need for a new layer of software that can help the designers cope with the tools in a CAD system. This new software is the “automatic design manager.” A design manager provides new forms of automation of the design activity: protection of the integrity of the design data, guidance for the novice designer, and advice for the expert.

The focus of an automatic design manager is directed more towards the design method-

ology than towards the pursuit of the design solution. “Design methodology” is the sequencing of the design tasks and the ordering in which the CAD tools are used, while the definition of “design solution” is left for now to the intuition of the reader.

The interest developed by the CAD community in automatic design management has strong industrial roots, namely the need to increase the productivity of CAD systems. But besides the utilitarian interest, design management also raises some genuinely theoretical questions regarding our understanding of the design process itself, as presented in the following sections and in the survey of previous work.

The breadth of this young discipline is such that the relevant literature is hardly bounded. Adjacent disciplines from which design management borrows freely include operating systems, data management, system modeling, and information modeling.

The system presented in this dissertation is a concrete proposal for an automatic design manager. A prototype of the system, implementing all of its essential features, has been developed, and it has been tested by many designers with various degrees of experience. This prototype is called VOV.¹ Sometimes the name “VOV” is used to refer to the abstract proposed system, rather than to the prototype.

This dissertation starts with an enumeration of the issues related to an effective use of CAD systems. An even superficial overview reveals that the issues are many and that they appear to be heterogeneous: automation of the design flow, bookkeeping of design alternatives, automatic documentation, coordination of team design, guidance for the inexperienced designer, and others. No issue has been discarded a priori, and an attempt has been made to provide a framework in which all these issues can find a natural setting.

VOV’s distinctive feature is that it does not attempt to *lead* the designers, telling them what is legal and what is not. Instead, the system *follows* the designers and keeps a *trace* of their activity. The trace is a graph that is used at once as the historical record of the design process, as a representation of the dependencies among the design data and as an executable program to sequence and schedule the tools. The trace is built automatically and non-intrusively with information generated by the tools at runtime. The server/client architecture used for the system is important to support multi-user designs and distributed processing.

The trace is the basis to provide many design management services: the consistency of the design data can be monitored and maintained, the design flow can be automated, concurrent tasks

¹VOV is also the name of an Italian liquor similar to a mixture of eggnog and rum, consumed preferably hot, in a cold winter day, and appreciated for its reinvigorating gusto.

Table of acronyms	
ASIC	Application Specific IC
CAD	Computer Aided Design
IC	Integrated Circuit
VLSI	Very Large Scale Integration
ADM	Automatic Design Manager/Management
DMS	Design Management System

Table 1.1: Acronyms used in this dissertation.

can be coordinated, and novice designers can be guided. The trace also integrates the notion of *measurement* on the design data, to provide other services such as tracking of design specifications and performance estimation.

Some issues that are often associated with design management are only indirectly addressed in this research. For example, there is no concern about management of human resources, about how to keep designers motivated and productive, how to schedule the work week, or how to complete a project on time and on budget. Nevertheless, an ADM can have an indirect but significant influence on these issues, if it makes the design activity faster, cheaper, and less frustrating.

The next Section contains some introductory comments on design and on the list of requirements for an automatic design manager. Chapter 2 contains a survey of previous work in design management. Chapter 3 contains a detailed description of the design trace and of its uses. Some implementation details are presented in Chapter 4, of interest to those who intend to implement their own trace-based design management system. Finally, the results obtained during the use of VOV in real designs can be found in Chapter 5. Appendix A consists of a tutorial introduction to VOV. A brief synopsis of the tools mentioned in the examples is found in Appendix B.

1.1 A characterization of electronic design

The services provided by an automatic design manager depend greatly on the interpretation given to the notion of design. The dictionary (the on-line Webster's 7th Dictionary) suggests that a design is "a mental project or scheme in which means to an end are laid down." This is enough to agree on the fact that design is the conjunction link between some goals and their satisfaction, but it says nothing on how such a link is generated.

The goals of design are usually described as a set of constraints that must be satisfied si-

multaneously. Some constraints define the desired behavior and functionality of the solution, while others may be more domain specific. For example, in the design of an electronic system, domain specific constraints may be the minimum frequency of the operating clock, the maximum dissipated power, or the range of operating temperatures. Some design problems are characterized by the availability of a *cost function* that can be used to rank the proposed solutions by their estimated cost. These design problems are referred to as *optimization problems*, because their goal is to find a solution that minimizes the cost function.

The availability of complex and powerful CAD systems adds a new dimension to design. Design is now more than just a search of a solution to a given set of constraints; it is also a search for a methodology to produce such solution. The methodology should be automated, in the sense of minimizing the manual intervention of the designers, and it should take full advantage of the power of the tools. Contrast this with Bushnell's specific definition of VLSI design [10]:

VLSI chip design is essentially a search process in the design space at the floorplanning level, to find a floorplan that will lead to a correct and reasonably compact integrated circuit layout. (page 48)

We believe that design is also a search in the "methodology space." The designer has to describe not only "what" he wants, but also "how" he is going to get it. In electronic design this amounts to choosing an appropriate sequence of tool invocations.

Complex designs are often partitioned into smaller designs, which in turn may still be too complex and require further partitioning. This process adds a hierarchical structure to design. Depending on the strategy used to traverse the hierarchy, one talks about different design styles: in *top-down* design the focus is on how the constraints for the top-level problem are converted into constraints for the subproblems, while in *bottom-up* design the subproblems are solved before the top-level problem. It is also common to hear the loosely defined terms *yo-yo design*, and *meet-in-the-middle design*, which prove that reality is always more complicated than we would like it to be, and requires some acrobatic balance between the two extremes of top-down and bottom-up design styles.

Design need not be a linear process; instead, it is often iterative, because many actions need to be repeated to refine and optimize the design data, and tentative, because many alternatives are often explored [31].

Particularly stimulating and provoking is Sequin's definition of design: "design equal documentation" [45]. Documentation constitutes a fundamental aspect of design, especially in the context of a company that wants to protect itself against the risk of seeing large chunks of know-how

depart together with its transient designers. Equally important is the role played by documentation in the context of team design, as a communication medium between the components of the team. Designers, however, often perceive documentation as a distraction from their pursuit of the design goals, and tend to avoid it; hence the need to provide tools to support automatic documentation of the design process.

Design is the act of conceiving a plan with some purpose in mind. A plan is the final product of design, not an input or a constraint. Thus, design is *goal-directed* more than it is *plan-directed*. The designers are going to do whatever is necessary to achieve their goals. They can adopt a predefined design methodology (a plan) as long as they are performing routine activities. But if their design has some new or unique feature that stresses the limitations of the existing design methodology, the designers will not hesitate to change their course of action to achieve their goal.

During the design activity, a plan can be a useful resource but only a weak one. This notion of a plan as a weak resource is borrowed from Alison Lee's research on the use of the history mechanism as a means of support for human-computer interaction [35]. Lee's attention is on short-term interactions, such as a session with a UNIX shell, such as `csh`. This type of interaction is found to have two properties: it is *recurrent*, because users frequently repeat their actions, especially the most recent ones, and it is *situated*, in the sense that the actions *are never planned in a strong sense* because *the circumstances of the actions are never fully anticipated and are continuously changing*. Lee suggests that a history facility is the appropriate resource to *relate knowledge and action*. Such a facility consists of four components:

The *collection* component records past user-computer interactions into a history. The *presentation* component displays the history. The *selection/modification* component allows the user to copy (and possibly modify) a history item. The *submission* component allows the user to *use* the selected history item in the context of the current situation. ([35], page 4)

The history facility has many uses: it allows the *reuse* of an history item to reduce the number of keystrokes; it offers *navigation* information by reporting to the user *where they have been and where they are*, or *reminders* on the status of the system; it provides *examples* on how some commands can be sequenced to achieved certain goals; it can be the foundation for some automation services, such as *macro facilities*, automatic *playback* of command sequences, *prediction* of the next user command.

Computer aided design is also a form of human-computer interaction that, just like Lee's interactions, is repetitive and situated (goal directed). To be accurate, it should be pointed out that

computer aided design is a *long-term* interaction and that it normally involves a *team* of designers, but neither fact affects the role of history as portrayed by Lee.

In the next chapter we show that there is a common thread that unifies most of the previous work in design management, that is the focus on planning, expressed by the attempt to capture the complexity of the design activity a-priori. VOV is the only ADM that focuses on the design history as the fundamental vehicle of interaction with the designers.

Design is not always a creative process. Often design can reliably develop along a pre-defined plan, following a well established *routine*. One such example taken from the *Octtools* is the design of a standard-cell circuit starting from its behavioral description: an effective methodology calls for the execution of three automatic tools, *bdsyn*, *misII*, and *wolfe*, with each tool allowing just a few options to tune its behavior to the specific circuit. This methodology can be described in a UNIX script that captures the known capabilities of the tools and that makes them readily available to the users. For a routine activity, a simple script can be a very effective design management tool.

If design was purely routine, the design management problem would have been solved in large part with existing techniques, such as scripts or other programs. The need for a more sophisticated system exists because, although a fraction however large of the design is normally routine, it is the remaining fraction that is the most challenging one, the one the designers need most help with, and the one that cannot be captured in an a-priori plan. Our experience with the *Octtools* confirms that all designs have had some unique feature that required a special and innovative solution: a new tool was needed, or the capabilities of some tool had to be stretched, or new testing strategies had to be invented. Unpredictability of the methodology is a characteristic of many designs.

Finally, we consider the role of computers in the design process, and in particular the new opportunities offered by computers in terms of management of the design activity.

VLSI design is probably the most extensively computerized form of design. Almost all the design information is stored in a computer memory, and most of the design activity consists of the execution of computer programs. This extensive use of computers in VLSI design has two notable consequences. First, it accentuates the iterative and tentative character of design by making it easier for the designers to explore many alternatives. Second, it makes it possible to obtain a great deal of data about the design process in a non-intrusive way, that is without disrupting the activity of the designers. Intrusive systems that require the designers to explicitly record their decisions and their actions, only add burden onto the designers and are often rejected.

One kind of information that can easily be collected is the *design history*. Computers

allow, in principle, the recording of every keystroke and mouse movement, as a way to capture the complete history of a design. A more economical approach is to record all the tool invocations, which is precisely what is captured by the design trace. The design history is an objective way to document the design, because it records what has happened, although it does not always record why it has happened.

In conclusion, the design history is not only a candidate to provide many useful services, as supported by Lee [35], but it is also easy to collect automatically. That is why the design history, captured by the design trace, is the focus of this research.

1.2 Requirements of an automatic design manager

An automatic design manager is a layer of software that assists the designers in their interaction with complex CAD systems. In this section we present a list of requirements of an ADM. The satisfaction of most or all of the items in the list is essential for the success of the ADM, that is for its acceptance in the users' community.

The first requirement is **user friendliness**, which requires an understanding of the users of the system. There is a variety of categories of potential users, and each category has different needs. Expert designers require a system that is powerful and non-intrusive, novice designers ask for simplicity and a high degree of automation, while project managers want to be able to exert control over the design process. Friendliness also implies a good user interface, a robust implementation, and a predictable behavior. Programmability of the user interface is important to give extensibility and power to the system.

Some may contend that the first requirement for an ADM should be a clear **model of the design activity**, complemented by a solid and effective **set of algorithms** that operate on the model. Models and algorithms are fundamental, but they require experimental validation, which is only possible with a strong user interface. Since an ADM is necessarily in close interaction with its users, its success is more dependent upon its user interface rather than upon its fundamental model or its algorithms. In fact, even good algorithmic solutions for design management can be made completely useless by a poor user interface.

An ADM should be **adaptable** and allow all existing tools to be easily integrated, because large and small corporations that may use the system wish to protect their current investment in CAD tools. The tool set may change because of the introduction of new versions of some tools, or of altogether new tools possibly coming from many different vendors, and the ADM should

effortlessly allow such additions and changes.

As the tool set may change, so may the **design methodology**. The ADM must continue to assist the designers especially in the uncertain times in which the design methodology is evolving. Established design methodologies should also be supported, by offering a language to describe them and a technique to apply them in a design. Support can take the form of non-intrusive advice, as favored by expert designers, or of strict enforcement, as preferred by the project managers.

The most challenging requirement derives from the large set of **services** that should be provided by an ADM. The services fall into one of two categories: protection and automation. A service provides protection if it prevents or corrects errors related to the consistency of the design data. An important protection service is the coordination of team design, or, more generally, the **coordination of concurrent activities**. When many designers work on the same design, it is necessary to coordinate the efforts of each person in the team to avoid conflicts, duplication of effort, or waste of time. Coordination of team design is easily achieved by a *locking* mechanism that gives privileged access to some design data or resources to one designer at a time. The locking mechanism is conceptually simple, and its effectiveness depends on its *granularity*.

Another protection service is **consistency maintenance**, which is based upon the notion of *dependency* among the design data. For example, the extracted view of a circuit is normally derived from the layout view, and therefore it depends upon it. If the layout is modified, the extracted view is no longer up-to-date and must be regenerated. The ADM should recognize dependencies between objects, it should detect when consistency is lost and it should be able to perform the appropriate actions to recover consistency.

A related service is the **bookkeeping of the design data**. For each piece of data, we want to know how it has been obtained and whether it is referenced in other parts of the design. CAD systems compound the bookkeeping problem, because they allow the exploration of many design alternatives, leading to an inflation in the number of pieces of data that have to be managed.

All the protection services are made more difficult by the heterogeneity of the design data. Data are often stored in different databases and accessed using different procedures. In the *Octtools* for example, there is a coexistence of *facets*, which are the storage unit in the database OCT, and UNIX files, either executables or data files. An ADM should not make assumptions about databases.

The basic automation service is the **automation of the design flow**, that is the automatic sequencing, scheduling and execution of CAD tools. The ADM should know how to invoke each tools, taking care of all the options and of all the input and output data. Upon completion, it should check whether the tool was successful. The value of an ADM can be increased by some **job con-**

control capabilities such as the dispatching of jobs to various machines on a local network, and the possibility of terminating jobs that are taking “too long,” maybe because they are stuck in infinite loops.

There should be no need to stress the importance of **documentation of the design process**, but this service is often overlooked. The ADM should maintain a detailed history of the design, by keeping track of what has been done, by whom, and when. The system should also allow free annotation of the design.

Another service is **assistance to novice designers**, whereby the ADM guides the inexperienced user through the CAD system and offers advice on what to do next to achieve a design goal, or on how to fix a problem.

Design estimation is the capability of predicting the outcome of a design methodology without actually running the tools; it is an important component in **design decision support**, for example because it can help the designers choose one methodology instead of another. Estimation provides tentative answers to questions about the performance of some tools, such as *how fast will this chip be if I use this tool?*, or about the design process itself as in *how long will it take to route this chip?* Any estimation is a difficult task in itself and is best performed by specialized tools. An ADM should however recognize the importance of estimation in design, and it should be able to provide the estimation tools with the information they need.

The **verification of design specifications** is an essential element in the design activity, because it determines whether the design goals have been achieved and therefore whether the design activity can be terminated.

Upon completion, a design is normally archived so that it can be recalled at a later time. **Archiving** can be expensive; it requires the storage of all the design data and of all versions of the tools used in the design, and possibly even of a copy of the current operating system, as a form of protection against software evolution that are not backward-compatible. It is the ADM’s job to indicate which data and which tools should be archived.

Finally, there is the important problem of coping with complexity: CAD systems may consist of more than a hundred tools, and some designs may produce many thousands of design objects. An ADM should be aware of this complexity and be efficient and responsive when dealing with **very large designs**.

1.3 Background on the *Octtools*

This research is based in large part on a CAD system developed at UC Berkeley and known as the *Octtools*. University software has some characteristics that make it a tough challenge for an automatic design manager, much tougher than the challenge represented by other industrial systems. University software is, first of all, the product of innovative research; its development often terminates once the research contributions have been made. Only occasionally the software is engineered and supported in order to provide an enabling platform for new research, or for educational purposes. University software tends to be relatively fragile, unsupported, undocumented, and rapidly changing, if compared with industrial strength systems. A design manager that successfully confronts all these difficulties in an imperfect CAD system, should stand a good chance in the world of commercial CAD systems.

Despite the historical symbiosis between VOV and *Octtools*, it should be emphasized that there is no dependency of one on the other and that the concepts in VOV are also applicable outside *Octtools*.

The next section is a brief introduction to the *Octtools*; it is included because some familiarity with this CAD system is useful to understand both the examples used in this thesis and VOV's implementation.

1.3.1 The *Octtools*

The *Octtools* are a set of loosely connected tools that operate with OCT, a system for data representation and storage management. [27]. OCT features a simple procedural interface and a general entity-relationship based data model that captures the information commonly used for the design of VLSI chips. Although OCT is not truly object-oriented in the C++ sense [48], it relies on the concept of data objects and relationships among them to represent the design. OCT objects have a *type* and a set of type specific *attributes*. The following objects are the most commonly used by the *Octtools*:

Facets: the facet is the fundamental unit for storing and manipulating design data in OCT. It has three main attributes called *cell*, *view* and *facet*. The *cell* represents the name of the design object described by the facet, for example it could be "2-INPUT-NAND". A cell can have many *views* to describe various aspects of it. For example the "physical" view can be used to describe the geometric layout of the cell, while the "logic" view describes

its logic behavior. A view can have several *facets*, which are different representations of the view. The *Octtools* use only two facets: the “contents” facet, which contains the actual definition of the view, and the “interface” facet, which contains an abstraction of the view. It is convention to refer to OCT facets with the notation `cell:view:facet`, although often the last attribute is not mentioned and we just say `cell:view`.

Instances: facets can be instantiated within other facets using the instance object. The instantiated facet is called the *master* of the instance.

Terminals and nets: these objects are used to represent connectivity information. A terminal represents a connection point for facets and instances, a net represents a connection between terminals.

Layers, Boxes, Paths and Polygons: these geometric objects are used to describe the physical layout of a cell.

Bags and Properties: bags are used to group objects, properties to annotate them.

OCT supports the relationship “contains” and “is contained by,” represented by a directed, unnamed arc. For example, in Figure 1.1 the net “X” is contained by the facet `buffer:logic:contents` and contains two terminals.

In order to allow flexibility of the data manager with respect to evolving needs in the design methodologies, OCT provides only a set of general *mechanisms* to allow creation, editing and retrieval of objects. This mechanism is independent of any particular design methodology or representation. Several *policies of use* have been chosen to specify how the mechanisms are to be used [32]. Adherence to these policies is the condition under which all of the *Octtools* can work on each other’s output. This is the key to the success of the system.

VEM is the users’ main channel of interaction with OCT because it provides the capability to view graphically and edit OCT facets. It also supports a remote procedure call protocol (RPC) that enables other OCT applications to register commands that can be invoked from within VEM.

A major feature of the *Octtools* is the capability to synthesize layout starting from a high level description of a combinational logic cell. The behavior of the cell is described using the language BDS. The program `bdsyn` translates the BDS description into a multi-level network of logic gates, stored in an ASCII format called BLIF. The program `misII` is used to optimize the BLIF description to minimize the area or the delay through the network. Depending upon the

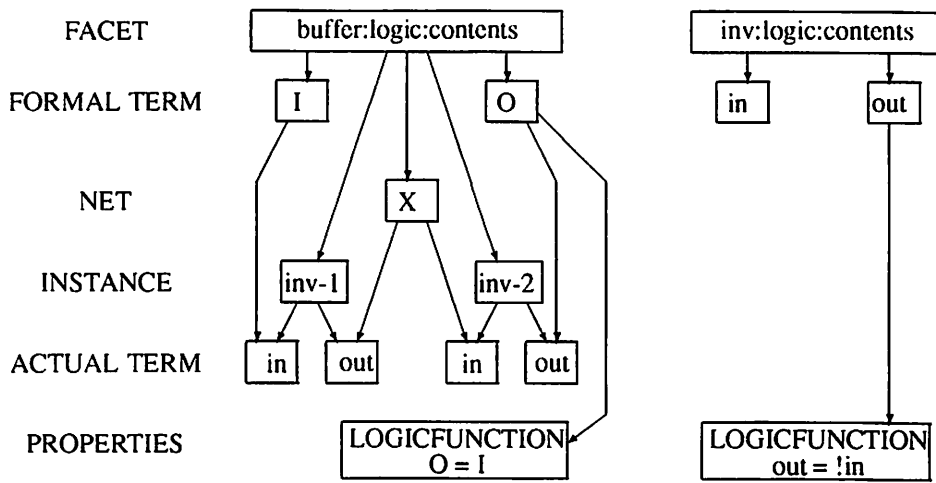


Figure 1.1: A buffer can be built as a chain of two inverters. The OCT objects that describe the buffer are shown here. The inverter is represented by the graph on the right: there are two terminals attached to the facet, and the output terminal contains a property that describes the logic behavior of the inverter. The graph on the left represents the buffer. The net “X” connects the output terminal of the first inverter to the input terminal of the second.

desired design style, `misII` will perform a *technology mapping* of the logic network. For example, if a standard cell circuit is desired, then `misII` will use a description of a standard cell library to implement the logic network using the gates available in that library. A netlist of standard cells produced by `misII` can be placed and routed with `wolfe`. Alternatively, `misII` can map the logic network into a more restricted library of gates that are implemented with GEM, a gate-matrix module generator. A PLA can be obtained directly from a BDS description using `octpla`.

The logic simulator `musa` is a multi-level simulator of combinational and sequential circuits. It performs switch-level simulation and also understands higher level models, such as buffers, RAM's and various types of latches. The typical usage of `musa` requires the preparation of a *simulation script* containing the commands to be executed as well as the tests that have to be verified.

The *Octtools* emphasize symbolic layout, which allows the specification of the topology of the cell to be decoupled from concerns about low level design rules. These concerns are resolved by the compactor `sparcs`, a tool that spaces the layout to minimize the area of the cell while respecting all the layout rules.

A subsystem called *Mosaico* is available to place and route macro-cell chips. A pad frame is built with *padplace*, and the core of the chip is placed using *puppy*, a program based on Simulated Annealing. Once a satisfying placement is obtained, *Mosaico* routes the chip by invoking a sequence of tools to perform channel definition, global routing, two-layer detailed routing, and via minimization. The resulting symbolic layout is finally processed by the compactor.

1.3.2 Brief history of VOV and the *Octtools*

In Berkeley, the *Octtools* are the third generation integration environment, built upon the experience gained with Ruby [19] in 1981-1982 and Squid [32] in 1982-1984. OCT, a revision of the Squid architecture, was available in prototype form in early 1986. The development of the OCT-based tools started in earnest in the spring semester of 1986, and proceeded at an intense pace for about two years. In the Spring of 1988, the tools were introduced in a VLSI design class at UC Berkeley, then taught by Prof. Randy Katz. This first semester was particularly hard on the students, because the tools were fragile and not sufficiently tested. As the tools became more robust in the course of the following semesters, more fundamental problems started to emerge, problems related to the interaction of the students with the tool set as a whole: students were oblivious of some tools, they did not seem to learn how to use some common and powerful options, they were forgetting to run the tools in the right sequence and they were destroying each other's data when trying to cooperate on the same project.

As a teaching assistant in the second semester in which the tools were used in the class, the writer had a direct relationship with the students and came to understand their difficulties. Also, as an author or coauthor of several of the tools that were being used (e.g. *musa* and the *Mosaico* set), the writer was well aware of how much potential was not being used, of how much effort is required to write CAD tools, of the extreme variability of user interfaces offered by the tools.

In Fall 88, we prepared some hand drawn "roadmaps," such as the one shown in Figure 1.2, to help the students understand the the design flow. These roadmaps were bipartite directed graphs, with nodes representing either data or tools. The students found this representation very useful and asked for more roadmaps. In January 1989 the development work on VOV began, as a way to automate the roadmaps. In the Spring semester 1990, VOV was introduced in an advanced graduate course on VLSI design, taught by Prof. John Wawrzynek. The eight students in the class cooperated in a design of a large chip for music generation called BRIC. VOV was used in conjunction with a large set of tools, including the *Octtools*, some commercial software such as

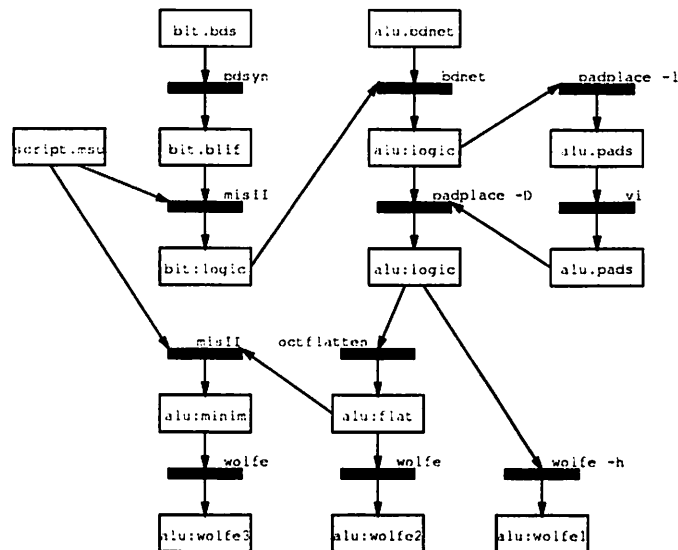


Figure 1.2: Roadmaps such as this are the precursors of the design trace.

Verilog, and some new tools especially written to address some of the peculiar problems in BRIC. The designers of BRIC provided a number of encouraging comments and suggestions on how to improve the user interface.

As the design of BRIC was proceeding, and the first feedback from real users was coming in, the VOV assistant took form. The first use of the assistant in a class was in the Fall 90, in an introductory graduate class on VLSI design. Once again, this class was taught by Prof. John Wawrzynek. The results of these experiments are reported in the Chapter 5.

Chapter 2

Previous Work

Electronic design management is a young discipline and still lacks a precise identity. It is also a broad discipline, including many issues relevant to other more specialized disciplines such as operating systems, data management, system modeling, office automation and user interfaces. The earliest work that can be considered specific to design management was published around 1985-86 [4, 16, 11, 30, 33], while the roots can be traced back to studies on system modeling, data-flows and Petri nets, in the seventies and before. Work in the neighboring field of office automation goes back to the late seventies, but it is only marginally relevant because of the substantial difference between office routine and design.

In the early eighties CAD researchers in Berkeley [41, 45] forecast the need to move beyond tool development and onto design management. But it was not until later in the decade, after the realization of complex CAD systems, that the need for design management became concrete and pressing, and researchers could move from abstract speculations on design management to proposals for practical engineering solutions. In Berkeley, the *Octtools* have played a key role in making this research possible. The system proposed here can be seen as a modern descendent of the `make.chip` utility first reported by Berkeley researchers in 1981 [41].

In this chapter, the contributions of previous work and the current trends in design automation are reviewed. This survey tries to avoid the tedium of a chronological exposition, preferring instead to present a few “cross sections” of the relevant literature, with each cross section looking at a particular issue in design management. The first and most substantial cross section looks at the fundamental problem of representation of the design activity. The second compares the intrusiveness of the proposed systems, the third analyzes the possibility of a taxonomy of design tools and design data. The trend towards the use of artificial intelligence techniques and some relevant issues

in data management are each given a section. Finally some commercial systems are looked at.

2.1 Representation of the design activity

The choice of a model for the representation of the design activity is a central issue in design management. With great generality, a CAD system can be thought of as a system of asynchronous concurrent processes, and the design activity can be thought of as the interaction between the designers and those processes. In this section we review the many models that have been proposed.

A preliminary key to interpret the various models can be found in an early study in the context of office automation. In [53] Zisman analyzes three models for asynchronous concurrent processes: finite state machines, partial orderings, and Petri nets. He finds that finite state machines are inadequate, because if the processes can be performed in any order the number of states grows exponentially with the number of processes. Partial orderings also lack modeling power due to inability to express the possibility that tasks should be performed one at a time but in any order. Zisman concludes that only Petri nets have sufficient modeling power and that both FSM and partial orderings are nothing else but restricted forms of Petri nets.

In electronic design, a consensus seems to be emerging towards the use of bipartite graphs or of Petri nets to describe the CAD transactions and their inputs and outputs [3, 34, 6, 13, 39, 49], but other models have also been proposed.

We propose a classification that separates the models that are specific to VLSI design from those that are more generally applicable to other forms of design. In the second class we find models based on directed graphs, others based on bipartite graphs, state machine models, and the blackboard model. Finally, a mention is given to those systems that evade this simple classification.

2.1.1 Ad-hoc models for VLSI design

Since its conception [20], Gajski's *Y-chart* model of the design activity in VLSI has been accepted with favor [21]. The Y-chart is not properly a model for asynchronous concurrent processes, but it has been used as a conceptual foundation to build such models. In the Y-chart, shown in Figure 2.1, three semiaxes in a plane represent all the possible states of definition of a VLSI system during the design process. The behavioral axis represents the degree of definition of the behavior of the circuit. The closer to the origin a point lies along this axis, the more complete the

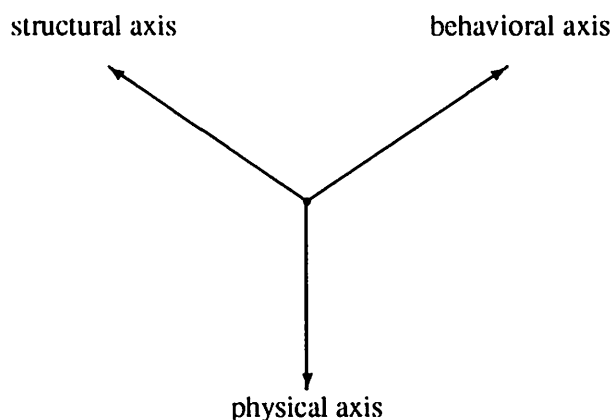


Figure 2.1: The Y-chart: a model for conceptual understanding of VLSI design, sometimes used also for design management.

behavioral representation. Similarly, the structural axis and the physical axis represent the degree of definition of the components used in the system and of their physical implementation. The design process is represented as a trajectory in the plane, moving from one axis to another and spiraling towards the origin, the point that represents the complete specification of behavior, structure and physical implementation of the system, and therefore the termination of the design process. The Y-chart has been used to compare qualitatively the performance and operation of various CAD systems and silicon compilers [9, 21].

Although not always explicitly, some management systems are based on the Y-chart and stress the importance of the trichotomy of design into behavioral, structural and physical domain. One example is Zimmerman's Playout [52, 46]. In Playout, the design flow is rigidly structured, because the system supports only a particular style of top-down design, on the assumption that such style leads to better designs and to fewer design iterations. CAD tools are grouped into six *toolboxes*, one for each step in the design flow: schematic entry, repartitioning, shape function generation (a step unique to Playout), chip planning, cell synthesis, and chip assembly. A central data manager provides permanent storage and supports communication between toolboxes, but each toolbox also has its own special data structures that are shared by all the tools in the toolbox. Each toolbox is a separate unit, typically a single UNIX process; it has its own *controller*, whose role is not clear, and its own user interface. A separate agent, the *Design Manager*, keeps track of the design process and

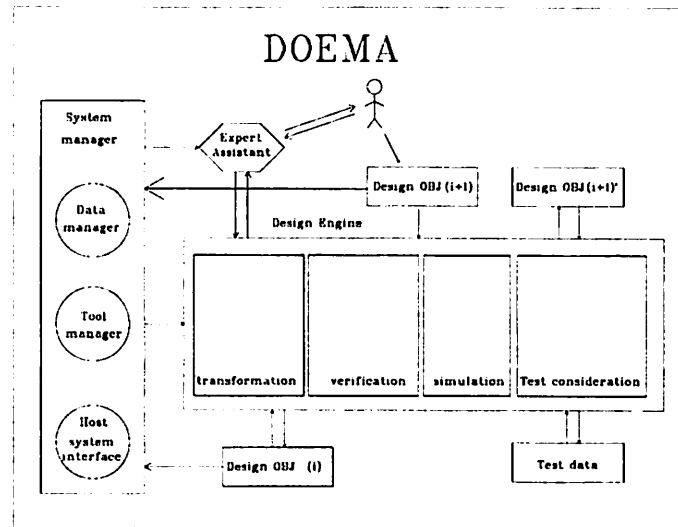


Figure 2.2: The DOEMA model.

directs the database manager.

The design flow is described by a bipartite directed graph, but the system internally does not use the graph. Some nodes in the graph represent the toolboxes, other nodes represent *data types*, which are either used or generated by the toolboxes. The graph is partitioned into three domains, behavioral, structural or physical. The toolboxes that span a domain boundary (e.g. behavior to structure) are those that perform a synthesis step. It is not clear if the graph represents the design history or if it is just a plan of how a design should be done. The notion of iteration, represented by cycles in the graph, is not clear. Parallel arcs represent alternatives, but there is no distinction between compatible or exclusive alternatives. Payout deserves attention more as a platform for tool integration and for the tools in its toolboxes than for its contribution to a conceptual understanding of VLSI design.

Another example of a speculative system is the conceptual framework for ASIC design proposed by Leung et al. [36]. Leung's goal is to bridge the gap between VLSI designers and VLSI technology, especially in ASIC design. Design is seen as a *decision making process*, which lies somewhere in the gray area between "Art" and "Science," between "creation" and "mechanical transformation." Modern CAD systems, Leung says, should first of all offer support for this decision making process.

The proposed *unified conceptual model* for ASIC design is shown in Figure 2.2. It is called DOEMA, as in Design Object, Design Engine, System Manager and expert Assistant, and it does

not meet many of the requirements for an ADM, most of which are also listed in [36]: simplicity, completeness, compatibility with existing tools, flexibility, ease of use. The model consists of three major units: the design engine, the system manager and the expert assistant. The design engine operates uniformly on all design object at all levels of abstraction. The main purpose of the engine is to invoke the appropriate tool to transform an abstraction of a design object into another more detailed abstraction. The engine also takes care of verification, simulation and testing. The system manager provides an integrated environment in which the engine operates, offering services such as tool management, data management and interfacing with the operating system, but the authors of [36] do not elaborate on the details of how such an environment could be implemented. The function of the expert assistant is to make the designer aware of all the possible alternatives, to both enable and help the decision making process. The assistant is an expert system based on two types of knowledge: one to determine the available alternatives and one to determine when alternatives should be considered.

DOEMA sounds like a designer's wish list: it would be nice to have a set of tools that operate uniformly at all levels of abstraction, a smart assistant and a versatile system manager. However, there is little or no indication that tool sets are about to become so well structured, and the claims about simplicity and versatility of the proposed model are not supported by any data. The DOEMA is a weak conceptual model of the design activity, and is of little use in the context of modern VLSI design.

Another system that emphasizes the decision making process in design is Yoda, by Dewey and Director [18]. The observation that early design decisions are the most important in terms of the performance of the final chip, leads Dewey to the notion of *Conceptual Design*, that is "the process of analyzing the outcome of alternative design decisions and their ramifications *before* actually undertaking specific design and fabrication steps." Conceptual design replaces the expensive trial-and-error approach of traditional design; its final product is a *design plan*, which is a set of *design decisions*, each consisting of a choice of one of the possible *design options* for each of the relevant *design issues*. Design decisions are related to each other by a set of ordering and consistency constraints. The designer is assisted in the decision process by a set of performance predictors and by advice generated by a rule-based expert system. Yoda is a particular instance of the proposed system for support of conceptual design, specialized to the design of digital filters, and it appears to be successful in its restricted domain. The generalization of Yoda to other forms of design appears to be difficult because of the challenges associated with the acquisition of the knowledge base and with the development of accurate prediction models.

2.1.2 General models for design activity

In this section we review some of the models that, although in large part originated in the context of VLSI design, do not rely on any specific characteristic of VLSI design, except perhaps the fact that it is already automated to a large extent. We review four types of models of the design flow. Two represent the design flow explicitly using either a directed graph or a bipartite directed graph. The others represent the flow implicitly, and they are the blackboard model and a state machine model.

Directed graphs

Knapp [33] reported some of the earliest works on VLSI design to give importance not only to the data but also to the operations of tools on the data. Knapp proposes the use of a bipartite acyclic directed graph for the representation of the circuits to be designed (the data), while the tool flow is represented by a simple directed graph, called *plan*; the nodes in the plan represent *abstract design states* while the arcs represent *abstract operators*. A design state consists of a set of assertions on the design, while an operator describes operations on the design state. An operator O is described by five entities,

$$O = \{F, E, Pre, Add, Del\}$$

where F is the executable code of the operator, E is a set of estimators for the operator, Pre is the set of assertions that must be true in order to allow the operator to execute, Add is the set of the assertions that become true upon execution of the operator and Del the set of assertions that become false. For example, the plan to produce a standard cell implementation of a combinational logic circuit starting from a set of logic equations consists of three nodes and two arcs, as shown in Figure 2.3. The first arc represent the logic minimizer, which requires the existence of a set of logic equation (precondition) to produce a set of 2-level logic equations (postcondition). The new state after the execution of the minimizer is described by the simultaneous existence of the initial logic equations and the new two-level form. The second arc represents the layout tool, which requires the existence of the two-level logic equations, an assertion that is true for the intermediate state in Figure 2.3, to produce a new state characterized by the existence of a standard-cell layout.

Knapp is aware of the difficulties related to the development of exact and complete descriptions of what each program does and favors instead a simpler description of the behavior of each operator based on the preconditions Pre , and of the postconditions Add and Del . The design

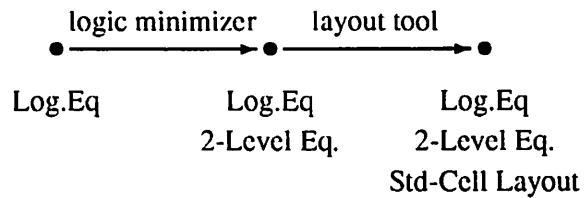


Figure 2.3: A plan produced by the DPE to transform a set of logic equations into standard-cell layout is represented by a directed graph in which nodes represent design states and arcs represent operators.

flow representation is both an executable plan and a design history; it is created as a plan, and it progressively becomes history as each tool is run. A Design Planning Engine uses artificial intelligence techniques to build a design plan tailored to each specific design. Such engine requires information provided by estimators that predict the performance of the various operators. These estimators need not be particularly accurate, but they should be at least *monotonic*, which means that for any given set of designs, the predictions should yield the same ordering as the corresponding actual performance measures.

One limitation of this model is that the notion of multiple inputs and outputs from a tool is not captured graphically, but through an overloading of the meaning of design state as an arbitrarily complex set of assertions. This can also lead to an exponential explosion in the number of states required to describe even simple notions, such as the existence of design objects. Consider a set of n design objects, each of which may or may not exist, independently from all the others. The assertion that an object exists can be either true or false, and the number of distinct states required to describe the existence of all objects is therefore 2^n .

Knapp's monolithic view of the design activity represents a case of premature pursuit of a totally automatic design system. His results are scarce, because the system requires idealized tools such as the "almost monotonic" predictors of the performance for each tool.

In contrast with Knapp's proposal, the Methodology Management System (MMS) developed at MCC [3] inverts the roles of nodes and arcs in the directed graph describing the design flow. In the MMS, nodes represent tools and arcs represent *dependencies* between tools. In most cases, a dependency represents a file that is created by a tool and used by another tool. In general,

```

(deftool bdsyn (bds-file blif-file &optional collapse )
  (version 1.1 )
  (tool-name "bdsyn" ) ;actual name of executable
  (doc "Behavioral synthesis tool. See 'man bdsyn'" )

  (args
; <value> <default> <description> <label> <use-flag>
  (nil nil "do not clean-up evaluation" "-b" nil )
  ((format nil "-c~A" collapse) nil "how much collapse to do" nil nil)
  (nil nil "suppress vector notation for one-bit values" "-o" nil)
  (nil nil "print table of non-assigned variables" "-n" nil)
  (nil nil "change SELECTALL's to SELECT's" "-s" nil)
  (nil nil "provide periodic updates of progress" "-u" nil)
  (nil nil "assign 'dont cares' to zero" "-z" nil)
  )

  (input-file bds-file nil "BDS file to be translated" " " )
  (output-file blif-file nil "BLIF format file" ">")
)

```

Figure 2.4: Example of Process-Tool definition in the MCC's MMS.

however, an arc can represent any abstract form of temporal dependency between two tools. Time-stamps are associated with each node and each arc in the graph. As shown in Figure 2.4, the graph is represented implicitly using LISP functions, with a LISP function associated with every tool in the tool set. Complex tasks can also be represented with LISP functions that combine various tools together. A separate tracing mechanism generates a textual log of the design activity, which is used only for documentation, and not, for example, to enable the replay of tasks.

By operating in a LISP environment, MMS can be extended and tailored to satisfy the particular needs of each design. But the need to maintain a detailed LISP description of the behavior of each tool opens the door to problems in the maintenance of the system. Compare, for example, the MMS description of the tool `bdsyn` shown in Figure 2.4 to the usage message produced by the tool itself and shown in Figure 2.5. The MMS description is a redundant repetition of information already easily available from the tool. It is also incomplete, because it does not mention the options `-d` and `-e` nor the legal arguments for the option `-c`, and it is slightly inaccurate, because it changes the meaning of the option `-b`. These are only minor flaws in the encapsulation of one tool, `bdsyn`, but imagine repeating the comparison for tens of tools and then remember that the tools are often changing; the result is a complex problem of consistency maintenance, between what the tools really do and what the MMS thinks they can do.

```

bdsyn [-bcdenosuz] [filename]
    -b      Turn off internal minimizing
    -c [n]  Specify less collapsing of logic
             0 No collapsing
             1 Local collapsing only
    -d      Give a dump of tokens during the parsing
    -e [n]  Specify execution level (0 ,..., 13)
             0 PARSE only           1 then dump
             2 through FOR          3 then dump
             4 through EVAL         5 then dump
             6 through LEAVE        7 then dump
             8 through VERSION      9 then dump
            10 through CLEANUP      11 then dump
            12 through LIF          13 then dump
    -n      Give information about unspecified variables
    -o      Omit the trailing <0> for 1-bit variables
    -s      Map all SELECTALL's to SELECT's
    -u      Give updates as to progress of execution
    -z      Set DONT_CARE's equal to zero

```

Figure 2.5: The usage message produced by the tool `bdsyn` is more detailed than the MMS Process-Tool definition.

A model that bridges the gap between directed graphs and bipartite directed graphs is described in the Task Manager by Chiueh, Katz and King [15]. Once again, an acyclic directed graph is used, but this time the nodes in the graph are complex entities with many ports. Each node represents a *task* and its ports represent the input and output data of the task. The ports are characterized by a number of attributes that specify, for example, the type of data and whether it is required or optional. Directed arcs connect an output of a task to an input of another task, as shown in Figure 2.6, so that complex tasks can be built by combining elementary tasks in a hierarchical fashion. This model fits in a four-layered scheme in which the entire *design process* is seen as consisting of several *design activities*, which are in turn composed by *design tasks*, which are a sequence of *tool invocations*.

Chiueh et. al. [15] embrace the notion that design is unpredictable, that it cannot be planned a-priori, but they do not carry this notion to its full extent. In fact, they restrict the unpredictability only at the level of a *design activity*, while they claim that design tasks can be planned. Within a task, the system knows at any time which tools can be invoked. The list of legal transaction is presented to the designer who chooses the one to perform next. The invocation of an unexpected tool within a task is possible, but it causes the system to react with a warning to the designer, al-

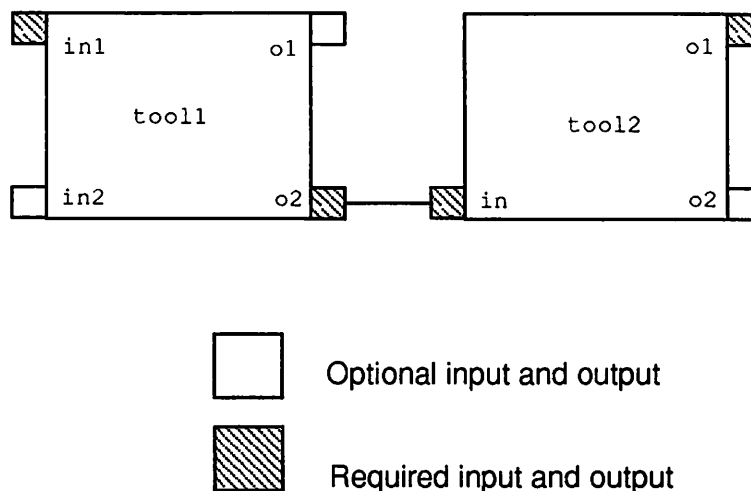


Figure 2.6: A node in the Task Manager is a complex entity, with one port for each input and output. Complex tasks can be described hierarchically by using arcs to join outputs of a subtask to the inputs of other subtasks.

though no other record of the exception is maintained. To support the design activities, the system maintains the *Activity History*, a possibly branching sequence of history records, each logging the invocation of a task and its inputs and outputs.

Within an activity, backtracking is provided by the possibility of storing *design points*, which are essentially copies of all the data that is relevant to an activity at a particular point in time. If the designer performs a task, and then decides to invalidate it, he can backtrack by asking the system to recover any of the previous design points. Although conceptually viable, this technique can become prohibitively expensive unless it is complemented by a sophisticated mechanism to reduce the redundancy in the storage of two adjacent design points, but there is no concern for this problem in [15].

The Task Manager, if implemented, could be vulnerable to the problem of using multiple representations for the design tasks: while the user interacts with the system by means of a graphical representations of the tasks, the system itself operates internally on LISP-like expressions. Multiple representations must be kept consistent and for complex CAD systems this might become an unwieldy job. Furthermore, there is the problem, already mentioned in the case of MCC's MMS, of keeping the task descriptions up-to-date with the tool set.

Bipartite directed graphs

The use of a bipartite graph to represent a design flow has been proposed by many independent researchers. In many cases the bipartite graph is an extension to the Petri net model.

Since their first appearance in 1962, in Carl Adam Petri's dissertation, Petri nets have been used extensively to model systems that exhibit asynchronous and concurrent activities. Many examples of such uses can be found in the excellent tutorial by Tadao Murata in the IEEE Proceedings [40].

Following Di Janni [30], a marked Petri net PN is defined as a set of five entities:

$$PN = \{P, T, I, O, M\}$$

where

$P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions

$I : T \rightarrow P^\infty$ is the input function

$O : T \rightarrow P^\infty$ is the output function

$M : P \rightarrow \{0, 1, 2, 3, \dots\}$ is the marking function

The places in a Petri net are conventionally used to represent data. The transitions represent operations on the data. The directed arcs represent the relations of inputs and outputs between data and operations. The marking function represents the number of *tokens* associated with each place, and $M(p) = k$ means that place p holds k tokens. The number of tokens in a place is normally interpreted as the availability of that number of the item associated with that place.

The behavior of a Petri net is characterized in large part by its *firing rule*. A transition t is said to be *enabled* if all of its input places contain at least one token. Only enabled transitions can fire, but the actual firing is determined by external factors not necessarily represented by the net. If a transition fires, one token is removed from each of the input places and one token is added to each of the output places. Situations of conflict arise whenever the firing of a transition disables another transition. If more than one transition is enabled, it is undetermined which one will fire first, but certainly one will fire before all the others, because simultaneous firings are not allowed.

Theoreticians have studied many behavioral properties of Petri nets, such as *reachability* of a marking from an initial marking, *boundedness* of the number of tokens held by places, and

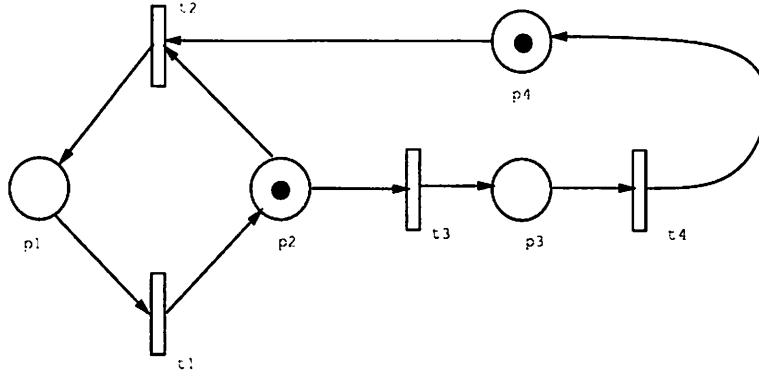


Figure 2.7: Example of a Petri net. Both transitions t_2 and t_3 are enabled. If t_2 fires, t_3 is disabled, and t_1 becomes enabled. If t_3 fires first, t_2 becomes disabled. This net is not live because all evolutions lead to a marking with one or two tokens on p_4 and no enabled transition.

liveness of a transition, that is the existence of a firing sequence that enables the transition. These properties have found no useful interpretation when Petri nets have been applied in the domain of design management, a circumstantial evidence that the Petri net formalism is perhaps more than necessary for design management purposes. An example of a Petri net is shown in Figure 2.7.

The first use of Petri nets in design management is found in the system *Monitor* by Di Janni [30] which is based on an extended Petri net with disabled transitions (DTPN) and with a special marking function. As usual, *places* represent files and *transitions* represent CAD tools. A *DTPN* is formally defined as a set of seven entities:

$$DTPN = (P, T, I, O, D, M, F)$$

where P, T, I and O are the same as in a marked Petri net and

$$M : P \rightarrow \{-3, -2, -1, 0, 1, 2, 3\} \quad \text{is the marking function}$$

$$D : T \rightarrow T^\infty \quad \text{is the disabling function}$$

$$F : T \rightarrow B = \{TRUE, FALSE\} \quad \text{is the firing function}$$

The marking function has an unusual range, the integers between -3 and 3 , instead of the set of nonnegative integers. The notion of negative marking was introduced in *Monitor* to denote obsolete objects. The other markings represent possible conditions of the files: 0 = non existent, 1 = provisional, 2 = final, 3 = confirmed. The disabling function maps transitions into bags of transitions. This could be represented by sets of arcs each incident on two transition nodes, thus

voiding the bipartite property of the graph. Di Janni, however, chose not to represent D explicitly in the net.

The **firing rule** used in Monitor is the following: *transition t_j may fire if the marking of all inputs p_i of t_j is greater than the number of arcs from the place to the transition, and if $F(t_j)$ is $TRUE$:*

$$\forall p_i \in P. (M(p_i) \geq \#(p_i, I(t_j))) \wedge F(t_j)$$

where the symbol $\#(x, B)$ is read as “the number of occurrences of x in the bag B .”

The new marking M' after transition t_j has fired is given by these rules¹:

$$M'(p_i) = \begin{cases} M(p_i) & \text{if } \#(p_i, O(t_j)) = 0 \\ \#(p_i, O(t_j)) & \text{if } \#(p_i, O(t_j)) > 0 \end{cases}$$

$$F'(t_i) = \begin{cases} F(t_i) & \text{if } \#(t_i, D(t_j)) = 0 \\ FALSE & \text{if } \#(t_i, D(t_j)) = 1 \\ TRUE & \text{if } \#(t_i, D(t_j)) > 1 \end{cases}$$

In words, the marking of an output place after execution of a transition depends on the number of arcs going from the transitions into the place. The net does not consume the tokens in the input places, as expressed by the fact that the marking of an input place is not affected by the firing of a transition. The disabling function D is designed to disable the tools that would overwrite the files that have already been created by other tools, but it can also be used for the opposite purpose of enabling some transitions.

Monitor has the notion of *consistency* of the design data and knows how to react when consistency is lost, for example when a disabled transition is forcefully fired by the user. Parallel paths represent alternative methods to obtain a particular piece of data. The choice of one path disables all others. The designer interacts directly with the net which appears in a window on the screen. An intelligent color coding scheme informs the designer on which transitions are enabled, by making them appear green, while disabled transitions appear red. The designer can always force the execution of transitions, even the disabled ones, but cannot change the topology of the net. The interface towards the tools is simplified to the point that the input is just the name of the working cell and the output is just a termination code.

Monitor has many innovative feature but it has a simplistic tool interface, it lacks explicit alternative management and it provides no way to maintain a history of the design. The operation

¹The equation for F' is a corrected version of the one in [30]

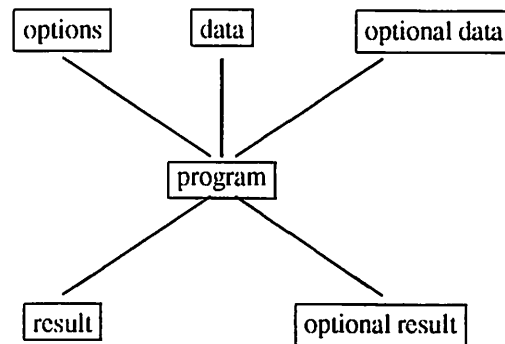


Figure 2.8: The basic primitives used by `decol` in the program environment templates.

of the system with a multiple cell design or with a hierarchical design is also not clear. Monitor has other weaknesses as an ADM, such as the rigidity of the Petri net, which cannot be changed by the designer, the monolithic architecture and the single-user interface.

Kozminski at the Microelectronics Center in North Carolina (MCNC) [34] has proposed a bipartite graph to represent *program environment templates*, each representing the set of all possible input/output relationships between a program and its data files. The system is called `decol`, for *Design Control Language*. The primitive entities used in the templates are shown in Figure 2.8. These templates, entered in textual format, provide a rich set of functions. For example, they can be used to rename program options so as to make them consistent with system-wide conventions, to rename input and output files, to define the rules that construct the command line, and to specify *simple actions* to perform before and after the execution of the program, although the notion of simple action is not well defined. The templates can also express complex interactions among different programs, such as the condition that the input of a program must be processed with a particular option by another program.

All the program templates are linked together in a *data flow template* that represents *all possible interactions and data flow paths in the given design system*. During a design, `decol` inspects the file system to look for files that match the characteristics of nodes in the data flow template. If more than one match is found for the same node, the data flow template is augmented with a new copy of that node and of some of the adjacent nodes, but the details of the augmentations are not clear. The file system is inspected again after the execution of any program. Since no details are given on the way this inspection is actually performed, it is fair to suspect that in the case of

large file systems, and for large designs, this inspection could be very expensive.

Several other constructs are available in `decol`, but they are separated from the program environment templates. For example, *goals* are groups of data files representing an objective that the user wants to achieve, *routes* are paths in the data flow template, *selectors* enable `decol` to choose one of several alternative subdesigns by some criteria that rank the quality of the alternatives, *overrides* modify the default behavior of a program, *comparators* compare different revisions of data files. Support for versioning is provided through an interface to the UNIX Revision Control System RCS.

By checking timestamps in the files, `decol` emulates the capabilities of the UNIX utility `make`. Some heuristics allow `decol` to bypass the execution of a program if it is unlikely that the output of the program will be different from what is already stored in the file system. This is attractive for the savings in processing time, but it is also an imprudent design practice, because it relies on weak heuristics to guarantee the consistency of the design data. The hierarchical structure of the circuit is embedded in the UNIX hierarchy of directories, probably a reasonable arrangement in many cases but in general restrictive and not justified. Perhaps the greatest limitation in `decol` is the lack of support for concurrent activities.

The work at Siemens by Bretschneider and Lager [6, 7] emphasizes the integration of Petri nets and *rules*. The nets describe the flow of information, the rules are used by an expert system to support design decisions and conflict resolution. This work is based on Predicate Transition nets (Pr/T nets) [23], an extension to the Petri net model that allows many types of tokens. In comparison with other models, these nets are remarkably complex. Transitions represent either tools or *decision nodes*, places represent either data or abstract *states* of the design, such as error conditions. Tokens represent the availability of design data, the occurrence of error conditions, or control information.

There are several types of arcs, each represented graphically by a distinctive symbol. One type of arc represents the *consuming access*, where a transition consumes one of its inputs. The *reading access* requires the presence of a token in an input in order to enable a transition, but the token is not removed when the transition is fired. Other arcs are related to the hierarchical structure of the design, and their meaning depends on the value of an *inscription* associated with the arc. For example, the inscription "all y" on an arc going from a place representing a netlist into a transition representing a schematic editor means that all the subcomponents of a module must have a netlist before the schematic editing of the module can begin.

Arcs are also labeled with *patterns* that influence the firing rule for a transition. The patterns identify the type of tokens that are required in the input places as well as the types of

tokens added to the outputs. A further complication arises because *variables* can be used in the patterns; in such case, the firing rule requires that all variables in all arcs entering and exiting a transition must be instantiated consistently.

In the Siemens system, the term *conflict* is used to refer to the choice among several alternative paths in the net. The final decision, which is responsibility of the designer, is assisted by a rule-based expert system. The decision point itself is explicitly represented by a transition associated with the relevant decision-making rules. Deviating again from the classical Petri net firing rule, this particular type of transition outputs different types of tokens depending on the actual decision. Decision nodes are also used to terminate design loops, such as optimization or testing loops. As with all decisions, the designer intervention is required to actually terminate each loop.

Entire subnets can be *colored*, where different colors represent different *scheduling policies*. For example, a scheduling policy may prescribe the enforcement of a particular design methodology (c.g. do simulation before layout), while another may simply require a passive monitoring of the designers' activity.

The nets are used as a purely descriptive device, which exploits the power of a graphical representation to give the users an intuitive understanding of the design flow. The nets are created manually by experts who edit them with the assistance of a smart graphical editor. In order to be used by the system, the nets have to be compiled into an executable format, that is into a set of rules that are then fed into a rule-based expert system. Each transition in the Pr/T net is converted into a set of three rules: a rule on the enabling of the transition, a rule to remove the tokens from the inputs when the designer decided to fire the enabled transition, and another rule to add tokens to the output places upon successful completion of the transition. The rule-base is complemented with the addition of other rules not derived from the Pr/T nets, such as rules about conflict resolution, parameter selection, and the special rules associated with decision nodes.

Perhaps the only problem with this model is its overwhelming complexity, which overshadows its remarkable modeling power. It is true that the real design world is complex and requires a proportionally complex management system, but a balance between detail and abstraction must be achieved to produce a system that designers can successfully use. The Pr/T net itself is difficult to read, even in its graphical form, and it loses most of its intuitive value because of the many annotations on the arcs and the many different tokens. Like other proposed models, this system requires a large amount of setup work to create and maintain the Pr/T nets and the other rules in the rule-base.

State Machines

State machine models of the design activity interpret each CAD transaction as a transition from a design state to another. The *NELSYS* project [37] has chosen this *state management* approach. The *state* of each design object is represented by the set of all transactions that have already been performed on the object, and it determines which new transactions can be legally executed on the object. The effect of each tool on the state of an object is determined by consulting an external rule-base.

Tasks performed by the state manager include checking of the starting conditions for each transaction and the possible automatic activation of pre-processing tools to enable a transaction for which the starting conditions would otherwise not be satisfied. The information required to perform these tasks is also stored in the external rule-base. The rule-base plays a key role in this system. Since it depends on the tool-set, it must be updated as new tools are introduced and old tools are modified.

NELSYS is an experiment in the development of a complete and sophisticated CAD system, including a substantial framework, open, efficient and configurable [50, 47]. The kernel of *NELSYS* is a configurable data management module based on the OTO-D model (Object Type Oriented Data model). Design management is based on the management of the *meta-data*, that is the data about the design activity. The meta-data represents the invariants of the design data organization rather than the actual composition of the design. The design management meta-data is organized on a per-project basis. A *Project Server* is in charge of managing the meta-data for the project, and clients connect to the server to query the meta-data.

The blackboard model

A different model of the design activity has been developed at CMU by Director and his coworkers. The model adopted by Bushnell and Director in the *Ulysses* system [11] is the *blackboard model*. The blackboard is a global database used to coordinate the activities of tools and designers. Each CAD tool is viewed as a **self-activating asynchronous** process, referred to as a *knowledge source (KS)*. The designer himself is a special KS. Knowledge sources communicate with each other via the blackboard. A KS is activated when a specified set of files is present on the blackboard. Any file modified by the KS is written back onto the blackboard. In *Ulysses* the blackboard is partitioned into three parts: the *CAD-tool blackboard* containing specific design data, the *Assertion blackboard* containing reasons for CAD tool failures, the *Scheduling blackboard*

containing the scheduling parameters for each KS. Resolution of conflicts among KS is performed by a special KS, the *scheduler*, implemented as a knowledge based expert system.

The detailed description of each design task, including the tool sequencing, is done within *scripts*, short programs written in the “Scripts Language” [10]. A script describes the tool sequencing, the motivation for each step in the sequence, the handling of exceptions, the rules for consistency checking, and other interactions between tools.

Aware of the evolving nature of CAD systems, Ulysses wants to offer an *opportunistic* flow control, that is a mechanism that permits the selection of the “right tool” depending on the available tools, and on other circumstances. By contrast, a *deterministic* flow control would instead use the same tools, as dictated by a pre-planned logic. But Ulysses fails to be truly opportunistic because of the scripts, which contain many explicit references to tools, and are therefore unusable when the tools becomes obsolete and are replaced by new ones.

Ulysses handles complex design dependencies, maintains data consistency and executes tools whenever some data change. Control directives from the designer are expressed as *posts to the blackboard*, such as “Run tool,” “Translate file,” “Place chip.”

The blackboard model has a definite appeal, stemming from its conceptual simplicity and from this idea of self-activating demons that come into play any time there is a need for them. However the blackboard model does little to simplify the problem of managing the design activity. Its simplicity stems from the fact that it pushes the complexity of the management problem into the knowledge sources, which must be smart enough to know when to become activated. The need for a *scheduler*, a super-KS capable of resolving conflicts arising between all the other KS, testifies to the inadequate modeling power of the model.

A more recent development of the blackboard model is Cadweld, developed by Daniell and Director [17]. The problems addressed by Cadweld are related to the interaction between designers and tools and to the introduction of new tools in a design framework. Designers often encounter difficulties learning new tools, they become discouraged and prefer to use familiar tools, even if they have become obsolete. Cadweld begins with a criticism of the *scripts mechanism* in Ulysses. In order to write a script, a designer needs a lot of knowledge about the environment and the workings of the system. The scripts make explicit references to tools, so that the addition of new tools require the modification of many existing scripts, and this complicates significantly the use and maintenance of the system.

Cadweld separates the information about CAD tools from the information about design tasks. Tools are once again seen as smart objects, capable of responding to certain posts on the

blackboard, but this time tools can only *volunteer* to perform a certain task. They respond to calls for action, but it is either the designer or a specialized program called *CAD task* to decide which volunteer is most appropriate. In this way, Cadweld eliminates the need for a scheduler such as the one in Ulysses, and it does so by placing the decision making burden upon the designers or the CAD tasks. A CAD task chooses a volunteer on the basis of some tool characteristics, such as its *robustness*, its *computing effort*, its *input list*, which are described by a CAD Tool Object (CTO). The CTO is like a *mushroom cap* on top of each tool, and hides the low level details on the invocation of the tool to simplify the interface between the tool and the CAD tasks. The *framework administrator* plays a decisive role in trying to keep up-to-date the CTO's, the CAD tasks, and the design rules that must be enforced by the system.

Another feature of Cadweld, its classification of tools, is described in Section 2.3.

2.1.3 Design environments

There is another category of design systems that take upon themselves some management tasks: the *design environments*, which we distinguish from the *design frameworks*. While a design framework is generally open to outside contributions, such as tools from external vendors, a design environment is a collection of tightly integrated tools. The value of a design environment is then measured by the value of the tools in the environment and by the ease with which new tools can be added. Design environments evade our classification based on the different models of the design activity, mostly because these systems tend to be monolithic, and not a system of asynchronous concurrent processes as assumed at the beginning of this section.

One such environment for VLSI design is SCHEMA, the work developed at MIT by Clark and Zippel [16]. SCHEMA is a software engineering experiment on object oriented programming, based on LISP and Flavors. Its goal is to simplify the development of synthesis and analysis tools by providing libraries of standard routines, by prescribing the use of uniform data structures and by providing libraries of advanced control structures deemed appropriate for CAD. Each design component in SCHEMA is a *module*, and consists of several descriptions, including one called the *topology* of the module, which represents its structure. The system tries to maintain consistency between the topology and the other descriptions by means of timestamps and *limited edit trails*, and the designer is warned if two descriptions become inconsistent.

SCHEMA, not the designer, defines which tools can be invoked and which cannot, although the designer is the agent that invokes the tools. The sparsity of the tools landscape is such that the

problem of tool sequencing and scheduling is not felt. The issue of coordination of concurrent activities is totally ignored, while data sharing among designers is treated superficially, simply by allowing a hierarchical organization of design objects.

2.2 Intrusiveness of implementation

By intrusiveness of an ADM is meant a measure of the extent by which the ADM changes the way designers do design. One form of intrusiveness consists of confining the users within the boundary of a prespecified plan. Although this is generally helpful for the inexperienced designer, it is too restrictive for the expert designer who may want to try a new tool or a new sequencing of the tools. If the system prevents a designer from violating the predefined plan, the designer has no option but to bypass the ADM, and may accidentally introduce chaos into the design process. Monitor, decol, MMS, the Siemens manager, all these systems fall into this class.

The unrealistic goal of a fully automated CAD system which relieves the designer of any need to make decisions is prominent in the literature. No system can claim to reach this goal. The systems that pursue the goal tend to be intrusive, because they assume a leadership role in their interaction with the designers. Designers are demoted to the level of tools, while the DMS claims the privilege to define what is legal and what is right or wrong ([11, 16, 15]).

For many of the proposed systems it is difficult to attempt an objective evaluation of their intrusiveness because the systems have not been developed enough to actually be used. Nevertheless, it is possible to estimate their intrusiveness from the published descriptions.

The most intrusive systems are probably those based upon the blackboard model. The whole design activity becomes a dialog between the system and the designers via requests posted on the blackboard, and the designers have no direct access to the tools.

Other systems require a special environment to be used. MCC's MMS [3] requires designers to work from within a modified Emacs editor and to understand a good amount of LISP code. Similarly, the Task Manager [15] requires that the design tasks be performed only from within a certain activity.

Many systems, including MMS, the Task Manager, and decol, tend to order an undisciplined set of tools by assigning meaningful and consistent names to tools and to their options, but the little gain obtained with this cosmetic change often has serious consequences. Some tools and some options that do not fit the clean and consistent naming scheme are ignored or eliminated. In the unspoken possibility that the design manager fails, the designers, who now have no choice but

to interact directly with the tools, will be forced to mentally switch back to the real names for both tools and options. The problem of consistent naming conventions should be solved at the root, by the tool developers and not by the design manager.

2.3 Classification of tools and data

Some design management systems require a classification of the tools, because they need to distinguish routing tools from placement tools, simulators from module generators. The purpose of such classification is to promote a better understanding of the tools set and to define groups of interchangeable tools. The danger is the possible explosion in the number of classes required to have a sufficiently refined classification. For example, the class of “placement tools” must be further subdivided into standard cells, macro cells, gate arrays, sea of gates and PCB tools, because they are normally not interchangeable. Similarly, a simulator can be logic, switch level, gate level, multi level, electrical, behavioral. The format of input and output data should also become a class discriminator, because two tools performing conceptually the same function are not interchangeable if they use two different input formats.

One classification hierarchy has been proposed by J. Daniell in Cadweld [17]. This hierarchy is a natural product of the object oriented approach used to describe the CAD tools, where a CAD Tool Object (CTO) can be derived from another CTO, inheriting all its properties. This derivation is represented as an arc in a tree, in which the nodes represent CTO classes.

The classification tree used by Cadweld includes more than thirty nodes, yet it is not sufficiently detailed to be useful as a design management device. This is shown explicitly in Daniell’s work itself, in a section that illustrates a typical session with the system. In response to a “post” on the blackboard requesting the placement of a standard-cell circuit, the set of tools that volunteered included the program `puppy`, which is specialized for macro-cell placement and is therefore totally inadequate for the job. Not only did `puppy` volunteer, but it was also selected and executed, only to conclude that it could not complete the job. Although one might always say that this type of mistake is attributable to an oversight in the development of a prototype, the example does show the dangers of relying on an unrefined tool classification.

However, it is not clear if any classification can properly capture the variety of all the tools in a CAD system. In the *Octtools* there are several tools that can perform many and different tasks. For example, `padplace` can be used to place pads or to route the power rings around a chip, while `misII` can be used to optimize of a logic network or as a format translator. Other

tools complicate any classification scheme, because they perform specialized and unconventional tasks. For example, `cprep -c` is called upon to patch a data structure produced by a faulty tool to enable the use of the data by other tools, `PGcurrent` annotates the power nets so that they will be properly sized by the routing tools. Some tools that are not traditionally regarded as CAD tools, can nevertheless be included in a design methodology, e.g. the UNIX utilities `diff`, `sed`, `cp`. The only classification that takes into account the variability of design tools has a majority of classes with just one representative, but such classification would be of little or no use, because it defeats the purpose of identifying interchangeable tools.

Some systems, including MMS, NELSYS and Cadweld [3, 37, 17], distinguish the tools that “do work” such as routers and simulators from the tools that perform “format translation,” with the implication that format translations are second rank operations. This distinction is a relic from the days of tool centered CAD systems [28], where each tool had its own input format and a large number of special purpose translators provided the fabric to integrate the tools into a system. Format translation is traditionally regarded as an annoyance and designers would rather not deal with it. Hence the tendency to separate translation tools from the other tools.

We see no conceptual difference between translators and other CAD tools. Translation can be conceptually hard, in the case of two languages which use different semantics, and it can be very time consuming. Like all CAD tools, translators also have inputs and outputs. Granting special status to translators is not useful and could be potentially dangerous when it hides from the designers the difficulties and the possible distortion introduced by these tasks.

`Decol` [34], the Task Manager [15] and the commercial EDMS [24] and the Integrator [43] have in common the notion of “conceptually different types of data.” This notion is often referred to as *strong typing* of the design data. The purpose of strong typing is duplex: it enables some safety checks on the inputs and outputs of tools and, as in the case of `decol`, it is used as a pattern matching criterion to automatically deduce a plausible design flow from a set of tool templates. The risk in strong typing is the same as in tool classification: once started, it cannot be stopped, and strong typing becomes unmanageable because of the large number of data types that must be considered. How large? In the *Octtools*, for example, following the logic suggested by Katz [15] or by Kozminski [34], one should consider at least the following sixteen types of ASCII files: `bdnet`, `bds`, `blif`, `esp`, `musa script`, `crystal script`, `mis script`, `cif`, `edif`, `spice decks`, `mask modification scripts`, `wolfe parameters`, `puppy parameters`, `puppy constraints`, `padplace lists`, `sparcs rules`. The same process for OCT facets would produce a similarly long list of types, the rule being that almost every tool in the set contributes a new data type.

Without detracting anything to the beneficial contributes of strong typing in other contexts, such as software development, strong typing for design management, although attractive, becomes unwieldy and probably unnecessary. In fact, the safety checks on the inputs of tools are redundant because most tools already recognize inputs of the wrong type by either explicitly refusing to execute or by simply failing. As far as matching of tool templates goes, type matching could be replaced, for example, by name matching.

2.4 Artificial intelligence techniques

A prominent trend in electronic design is the application of artificial intelligence techniques in the development of CAD tools. The same techniques have also been applied in design management, partly because the design process is heuristic and partly because an expert human designer, given enough time, can outperform the most sophisticated CAD tool. Researchers have tried to capture the designers' expertise in a set of rules used to drive an expert system. However, to date, these expert systems have yet to prove that their value offsets the efforts necessary to acquire and maintain the knowledge base.

One example is Steele's work at NCR [38] on a prototype expert system to provide advice on performance, testability and quality for standard-cell designs. The knowledge is acquired through interviews with several designers, including both experts and novices. Steele recognizes that the applicability of his system is limited to domains which are well understood and relatively stable over time, two characteristics which are rather uncommon in CAD.

Other systems have already been mentioned. One is the expert system proposed for the DOEMA framework [36], which assists the designer by alerting him about the available design alternatives. The Siemens system [6] is also rule-based, where the rules are in part derived from a Petri net description of the design flow and in part added to resolve some specific decision problems. Yoda [18] indicates artificial intelligence as best suited to control the *part of conceptual design that involves ill-defined knowledge*, while imperative programming languages are more appropriate for the well-defined part of conceptual design. The rule-based expert system in Yoda provides two forms of assistance: advice about the available alternatives, and prediction of the performance of an alternative. In the domain of digital filtering, Yoda offers a powerful design environment complete with an extensive set of accurate prediction models.

2.5 Issues in data management

Traditionally, the problem of efficient data management has been studied in its own, with emphasis on the data alone. The same problem becomes more interesting in the context of design management, in which both *data* and *transformations* on the data can be managed together. In particular, the notions of **consistency maintenance** and **version** should be reconsidered.

Batory and Kim [5] are mainly concerned about the problem of minimizing storage redundancy among versions. They propose a method based on *version-derivation hierarchies* that minimizes redundancy while maintaining uniform access to any version. While this problem of redundancy minimization is orthogonal to our current focus on design management, relevant is the problem of *change notification*, also considered in the same paper. Change notification for Batory and Kim is what could also be called *consistency management*, that is the problem of defining and automatically maintaining a notion of consistency between design data. In particular, Batory and Kim look at how changes in a particular design object should be propagated to other objects that reference it.

Two techniques are proposed, one based on timestamps, and another that uses auxiliary directories to store a log of all changes. With the first technique, each design object is given two timestamps: a *change-notification timestamp* (CN) that indicates the last time the version was changed, and a *change-approval timestamp* (CA) that indicates the last time a designer has approved the latest changes. The consistency criterion requires that $CA \geq CN$, for each version V as well as for each version referenced by V .

Change notification can be active or passive. In passive change notification, the designer is notified of a change in a version the first time that version is referenced, while in active change notification the system notifies all interested designers about a change right after the change happens.

Batory and Kim talk about the possibility that a *small change* in an object A might have no effect on an object V that references A . In such a case, the notification mechanism should stop at V , because the objects that reference V need not be notified. Such a selective notification strategy requires less work than a blind *notify all* strategy. The objection to this approach is that the notion of “small change” has no formal definition, that could be used reliably in an automated CAD system. Batory and Kim can afford to be informal, because their definition of consistency is based on CA , that is it requires the intervention of a designer who makes the decisions and assume responsibility for them.

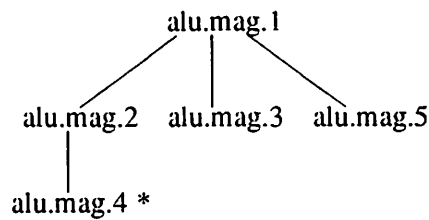


Figure 2.9: A version derivation tree in the Version Server. Nodes represent versions of the layout of an ALU, arcs represent the relation of derivation of a version from another. Version 4 is marked as being the *current* version, even if it is not the most recent one. What is missing is an indication of how, and why, each derivation has occurred.

In the Version Server described by Katz et al. [31] the design objects are arranged in a space described by three axes: composition, derivation and equivalence. The objects are also grouped into **workspaces**. A workspace operates as a metaphor of an electronic file cabinet, in which objects are checked-in and checked-out according to some rules. In a typical design environment there are three types of workspace, which are distinguished by their check-in and check-out policies. *Public archives* have a liberal check-out policy but a restrictive check-in policy, *private workspaces* have a liberal check-in procedure and a restrictive check-out one. *Group workspaces* lie somewhere in between archives and private workspaces and allow cooperation between designers. Workspaces offer some protection of the integrity of the design data, because the important data are kept in the public archives, where the restrictive check-in procedure prevents accidental corruption of the data. However, within each private workspace, the designer is dangerously free to corrupt the design data.

The relationship among the versions of an object is represented by a *derivation tree*, such as the one shown in Figure 2.9. The nodes in the tree represent the various versions, while the arcs represent a generic relation of derivation of a version from another. Each derivation tree has a *currency indicator* which points to the version “of interest,” which is not necessarily the newest. The derivation tree contains no mention of how a new version was derived from another one, nor why.

There are many data managers and they each have different capabilities. For example, Frank Halasz in his work on the next generation of hypermedia systems [26], insists on the need for a rich versioning scheme, like the one adopted in PIE [25]; not only each entity has its own version

history, represented as a linear graph, but also whole sets of entities can have a version history, which is used to keep track of coordinate changes to the objects in the set. For efficiency reasons, Halasz recommends that a data management system allows a representation of both the versions and the “deltas” between versions, rather than just the versions.

Since not all data managers have the same capabilities, it is important to try to decouple the ADM from the capabilities of the data manager. In electronic design, despite an intense effort by CAD developers to integrate all data management into a single database, the integration is far from complete. For example, in the *Octtools* there are at least two databases: OCT for the management of all layout, netlists, and schematics, while UNIX is used for the management of ASCII files and executables. It is restrictive, to assume that all the data are homogeneous and managed by the same database, but this is commonly done. For example the NELSYS system [50] manages only data stored in its data manager (the sophisticated OTO-D), as does Playout [52], while Monitor [30] deals only with UNIX files.

2.6 Commercial systems

The demand for automatic design management has recently become strong and a number of players have appeared in the arena to compete for a share of the electronic design market. There is probably no market for stand-alone design frameworks, because the value of a framework without tools is minimal. The most important part of a CAD system remains the tool set and users expect the framework, and its services, to come with the tool set.

The Electronic Design Management System developed by EDA [8, 24] is probably the most famous ADM available. EDMS appears to meet all the requirements of a powerful ADM: it is programmable, expandable, customizable, and it offers a complete collection of services, including protection using workspaces, consistency management, history tracking, and maintenance of profiles for each user authorized to work in a workspace. But all this is offered in the absence of a clear model of the design activity.

One goal in EDMS was to remedy some of the shortcomings of standard operating systems when applied to electronic design. In particular, the protection and versioning schemes of conventional file systems were perceived as inadequate, because they offer only simple version chains instead of alternative version paths. Another goal was to provide a framework for the incremental evolution of CAD systems, by allowing the integration of tools produced by different vendors.

EDMS consists of three major parts: the Workspace server, the Application Run Time System and the Desktop Shell. The workspace server implements most of the capabilities described by Katz et al. [31], including team design support, versioning with derivation trees, and the notion of *current* version. Programmability of the framework is possible with an interpreted version of C, called E-language, that is used for both tool encapsulation and to allow the user to specify design policies that can be enforced by the Workspace Server. Some tasks are triggered by automatic check-in/check-out procedures; others are invoked explicitly by the designer.

The system requires a complex and time consuming tools encapsulation [24]. The capsule for each tool must describe all required and optional inputs and outputs, including their types, description of the tool itself, pre-conditions and post-conditions. As reported in [24] the encapsulation of the program HSPICE took three days to plan and seven days to write.

No particular assistance is offered to the inexperienced designer, other than through a sophisticated menu-driven and graphical user interface. The interface represents the tools with a slot for each of their required and optional inputs and outputs. The user fills the slots with the appropriate data and then asks the system to run the tool.

Another example of design framework with management capabilities is the Integrator by Interact [43]. The system consists of a collection of sophisticated procedures for tool and data management wrapped in a powerful graphical user interface. The system emphasizes distributed processing and cooperation among designers, and provides a mechanism for active change notification. Design management is based on a static encapsulation of each tool's inputs, outputs, and other characteristics, entered through the graphical interface. Strong typing of the data allows automatic generation of tool sequences.

2.7 Conclusion of the survey

The systems presented in this survey are unsatisfactory as design management tools for one or more of the following reasons.

- Implementation difficulties, particularly in rule-based expert systems like Yoda [18], which face the challenging problem of knowledge acquisition.
- Intrusiveness.
- Poor quality of the final designs, due to a rigid interface to the tools, which prevents the designers from having full access to the power of the tools.

- Incomplete set of services.

Most of the literature on automation of design management assumes that the design process can be planned, and that it makes sense to try to *write a program* to capture the complexity of the design activity a-priori. The preconditions, postconditions in Knapp's description of the operators, the Petri net in Di Janni's Monitor, the scripts in Ulysses, the characterization of the CAD-Tool-Objects in Cadweld, the rules in NELSI, are all different ways of programming the design activity. Such programs have been found to be extremely difficult to develop beyond the stage of a prototype.

In Section 1.1, it is observed that design is a goal-directed activity that is not apt to be confined within the boundaries of a predefined plan. The difficulties encountered by the developers of the systems reviewed in this chapter are probably related to their approach of focusing upon an a-priori description, rather than on the history of the interaction between the designers and the CAD system.

Chapter 3

The Design Trace

In the previous chapters we have presented the design management problem, highlighted some technical issues, and we have surveyed previous work to identify, by contrast, the innovative contributions of this work. In this chapter we describe a design management system based on the notion of design traces. Most ideas presented in this chapter have also been implemented in a prototype called “VOV.” The name VOV will also be used as a short-hand to refer to the proposed trace based management system.

This chapter begins describing the design trace. In Section 3.4 the server/client architecture of the system is shown. Sections 3.8 and 3.8.1 introduce the notion of sets and of how they are used to represent hierarchy in the trace. Section 3.9 is dedicated to the services provided by the system. The notion of measurement and its uses are presented in Section 3.10. A most important service is assistance to novice designers, as provided by the VOV assistant, a program that extracts information from a library of example traces, all described in Section 3.11. The notion of iteration in design. is presented in Section 3.13. The principles that have been most effective in the development of VOV are summarized in the final Section 3.14.

The next Chapter 4 contains a description of some implementation details of the system.

3.1 Design management based on design traces

The goal of this and previous research is to propose an ADM that adds value to current CAD systems by making them easier to use and more productive. However, this research differs from the previous, because it follows a different conceptual itinerary to achieve the goal. Our basic idea is to **begin** by building a system that does not try to *lead* the designers through a predefined

plan, but a system that *follows* the designers, by monitoring and recording their activity. The record of the design activity is represented as a bipartite graph, called the *trace*, which is used as the historical record of the design, as a way to capture data dependencies, and as a device to direct the automatic execution of tools. The trace captures each tool invocation and registers all inputs and outputs of each tool. This tracing mechanism is meant to satisfy the needs of the expert designers; it provides services such as consistency maintenance and coordination of concurrent activities, while maintaining unrestricted access to the tools.

Once the tracing mechanism is in place, the capabilities of the system can be extended to include services such as guidance and assistance for novice designers, and support for design methodologies. These extensions are based on the analysis and reuse of the design traces.

The trace is not just a “model” of the design activity, it is also a “machine.” A model is an abstract representation of some features of a system, mostly used to do theoretical studies. The trace is not only an abstract representation; it is also used to monitor, represent and automate the design activity. Like all machines, the trace is used to modify reality.

3.1.1 Non-intrusive tracing

The trace should be captured in a manner that is as non-intrusive as possible, requiring no effort from the users. One solution, the first considered but then soon discarded, is to provide a *shell* in which the designers operate. The shell would feel like a regular UNIX shell, e.g. `csh`, but it would also know about the CAD tools and about their effects on the design data. The DMS would maintain the trace using the information generated by the analysis of the command lines intercepted by the shell.

Such a shell would be too difficult to realize. First of all, it would require an immense amount of knowledge, which would have to be maintained up-to-date with a rapidly changing tool set. But, more fundamentally, no shell can predict the behavior of a tool from an analysis of its command line, no matter how sophisticated the analysis. For example, the command line

```
misII -f script.msu file.blif
```

does mention a file called `script.msu`, but it does not say that the file could be either in the current directory or in

```
~octtools/lib/misII/lib
```

nor does the command line show that the file

```
~octtools/lib/misII/lib/script
```

is also used as an input – it is referenced by a command contained in the file `script.msu`.

This digression is to prove that the rules to determine inputs and outputs of a tool can be arbitrarily complex and are not derivable from an analysis of the command line. The only agent that has complete knowledge of what a tool does is the tool itself, at runtime. This leads to the principal assumption in VOV: *the tools themselves, at runtime, generate the information used by the system to maintain the design trace.*

The system has no direct way to know whether the information provided by the tools is reliable. The system has no choice but to trust the tools, even if this might sound dangerous. In practice, this has not been a problem. In section 3.4.1 we show two techniques that allow tools to produce complete and correct information.

3.1.2 The trace

The trace is represented as a *bipartite directed acyclic graph*, sometimes called *BAD graph*, similar to a Petri net, but simpler. From Petri nets we borrow some key terminology: the nodes in the trace are either *places* or *transitions*. Each **place** represents a generic piece of design data, for example a UNIX file or an OCT facet. Each **transition** represents an atomic transaction that can be initiated from a UNIX shell, such as the execution of a placement tool or a logic simulator. Arcs express input/output relationships between places and transitions. An arc from a place to a transition indicates that the place is an input for the transition; an arc from a transition to a place indicates that the place is an output.

A place represents data of different types: ASCII files, executables, OCT facets, and others. In the graphical representation of the trace, different icons are used for each type of place, as shown in Figure 3.1. All transitions are instead represented graphically by the same icon, shown in Figure 3.2. The direction of each arc is not represented explicitly, but in our representations arcs will always be directed downward, towards the foot of the page.

The main attributes of a transition are its *working directory* and its *command line*. The input/output lists of each transition are computed at runtime, and nothing can guarantee that, for

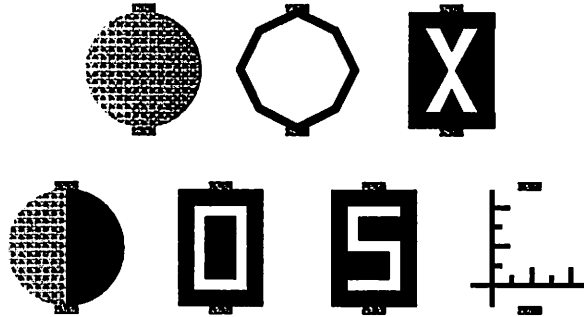


Figure 3.1: Places of different type are represented by different icons. Left to right, in the top row we have: ASCII files, OCT facets, and executables. In the second row: booleans, command line options, exit status, and measurements.



Figure 3.2: All transitions in the trace are represented by this icon.

a given command line and a given working directory, the lists will remain the same each time the transition is executed. This prompts the following taxonomy of transitions, consisting of three classes:

Data-invariant: these are transitions that, when executed in the same working directory, always produce the same set of inputs and outputs; for example, the transition

```
cp file1 file2
```

which copies `file1` into `file2`, always declares that `file2` is an output and that `file1` is an input.

Data-sensitive: these are transitions that look at the value of some of the input places to determine other inputs and outputs. The classical example is a netlisting transition, such as

```
bdnet netlist_file
```

in which inputs and outputs depend on the value of the file `netlist_file`.

Pathological: these are transitions that generate different input/output lists every time they run; it is easy to construct a pathological transition, but only rarely would such transition be useful. There are CAD tools that have a pathological behavior because, due to excessive zeal, they create a new output rather than overwriting data that already exist. For example, consider a tool that on its first run produces a file, say `f1`. If the tool is invoked a second time, with an identical command line, the tool sees that `f1` already exists, and, instead of overwriting it, the tool produces the same file with a different name, say `f2`. Upon a third invocation, the tool outputs `f3` and so on. For the purposes of design management, such a tool is pathological and it has the annoying characteristic of cluttering the working directory.

Even a data-sensitive or a pathological transition may invariably declare some places as inputs and outputs. We call these the *essential inputs* and *essential outputs* for data-sensitive and pathological transitions. VOV handles all classes of transitions.

Both transitions and places are **nodes** in the trace. For each node we can talk about its inputs and outputs. The notion of input and output of a transition descends directly from the definition of trace. In the case of a place, the input, if it exists, is the transition that generates the place, and the outputs are the transitions that use the place as their input. A node with no inputs is called a “primary input.” Although it is conceivable to have a transition with no inputs nor outputs,

for practical purposes related to the scheduling of transitions, each transition must have at least one input and one output.

Each place can have at most one input, that is it can be the output of at most one transition. This is referred to as the *single assignment property* of the trace, a term derived from the interpretation of the trace as a definitional language, as described in Section 3.2.

For a node n , its input set is denoted by $I(n)$ and its output set by $O(n)$. The outputs of a node are said to “depend” on the node. The relation of dependency is transitive and the transitive closure of the nodes that depend on a particular node n is denoted by $D(n)$. If m depends on n then $m \in D(n)$ and there exists a directed path between n and m , denoted by $P(n, m) = \{n_0, n_1, \dots, n_p\}$ where $n_0 = n$, $n_p = m$ and $\forall i \in \{1, \dots, p\}, n_{i-1} \in I(n_i)$. The path need not be unique.

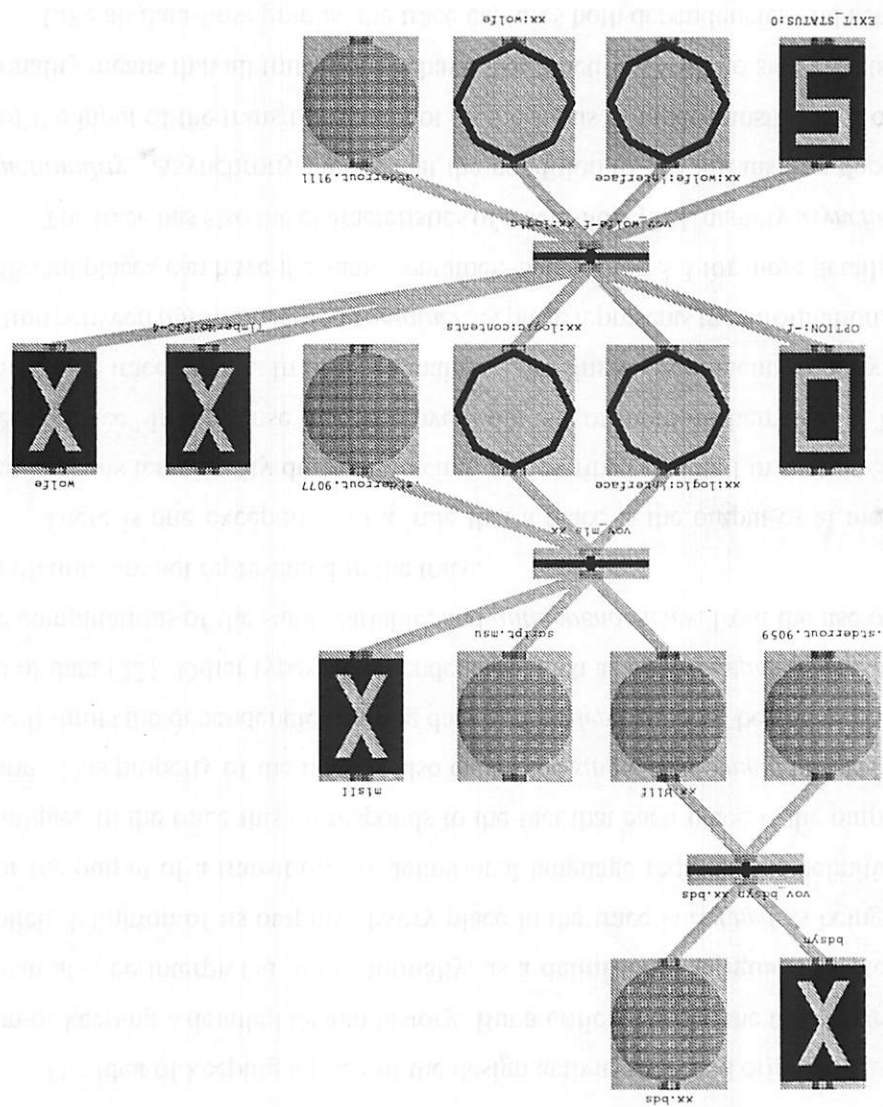
An attribute of each node is its *status*, which can take one of the following values: VALID, NOT VALID, DEAD, TRACING, RETRACING, MISSING. For the time being, we only need to be concerned about the two most common values, VALID and NOT VALID, while the other values are described in Section 4.1. A VALID node is a node that is up-to-date; it is either a primary input or the output of a successful transition, which is itself VALID. If a node n is modified, all of its dependent nodes $D(n)$ are no longer up-to-date, and they are marked as NOT VALID. A transition can be fired if all its inputs are VALID; if it completes successfully, all its outputs and the transition itself become VALID. According to these rules, the trace can also be interpreted as a data-flow graph.

We call **retracing** the firing of a transition that is already in the trace. It is different from tracing, which corresponds to the creation of a new transition in the trace. Automatic retracing is an important service provided by VOV, as described in Section 3.9.3.

3.1.3 An example trace

Figure 3.3 shows a trace left by the execution of three *Octtools* that transform a textual description of a combinational logic circuit into a standard-cell layout. Starting from a description of the behavior of the circuit, in the file `xx.bds` in the top row, the tool `vov_bdsyn` has been invoked to obtain an expanded set of logic equations (file `xx.blif`), which have been optimized by `vov_misII`, which also has mapped the optimized logic into a library of standard cells. The optimizer has produced the OCT netlist `xx:logic:contents`. Finally `vov_wolfe` has been used to place and route the circuit producing the OCT facet `xx:wolfe`. All tools have other accessory

Figure 3.3: Traces should be read from top to bottom: inputs are on top, outputs on the bottom. Three transactions are represented in this trace. Although all the places are treated similarly, they are represented graphically by different icons depending on the type of the place.



outputs, including a file to store `stdout` and `stderr`.

3.2 The trace as a definitional language

The idea of keeping a trace of the design activity emerged originally as a solution to the problem of keeping a detailed design history. But a critical look at the trace reveals that the BAD graph can also be interpreted, more formally, as a **definitional language** [1]: every transition is an implicit definition of its outputs. Every place in the trace is *defined* as being either a primary input or the output of a transition. A definitional language requires the definition of each entity to be unique. In the trace this corresponds to the fact that each place is the output of at most one transition. This property of the trace is also called the *single assignment property*. It is important, because it limits the dependencies among data to *flow dependencies* between the computation and the use of data [22]. Other types of dependencies, such as *output dependencies* between two successive computations of the same variable, and *antidependencies*, from the use of some data to its recomputation, are not represented in the trace.

There is one exception to the rule that a place is the output of at most one transition, and that happens temporarily during retracing, as described in detail in section 3.9.3. Some tools operate “in place” in the sense that they overwrite one or more of their inputs. These tools stress the limit of the trace model. In order to maintain the single assignment property, VOV requires a distinction between *information* and *container*. A place represents the information, not its container, and different places can have the same container. See section 3.3 for more details.

The trace has also the characteristics of a data-flow [22], namely *asynchrony* of operations and *functionality*. Asynchrony means that the condition to fire a transition depends only on the status of the input of the transitions and not on the status of other transitions or on a global clock. Functionality means that all transitions behave like functions, with no side effects.

Like all data-flow graphs, the trace captures both dependencies and scheduling information. Every transition in the graph is a **declaration of dependency**: all the outputs depend on all the inputs. The main difference with ordinary data-flow graphs derives from the fact that all dependencies are computed at runtime and nothing can guarantee that they will remain the same for every execution, despite the fact that normally they do.

The interpretation of the trace as a declarative language implies that the management of the design trace, and design itself, is a process of **refinement of a set of definitions**. Each time a designer executes a transition, he is effectively saying: I want the outputs of this transition to

depend on the inputs, and consistency is obtained by successfully completing this transition. If a place was already defined as the output of another transition, the system detects a conflict and requires the designer to confirm the new declaration. More about conflict detection can be found in section 3.9.4.

3.2.1 Backtracking

The interpretation of the trace as a definitional language has an impact on the notion of **backtracking** in the design process. Backtracking is often mentioned in the literature with expressions such as “exploration of new alternatives” followed by “backtracking to an old alternative.”

Cadweld [17] associates backtracking with its concept of design iteration, that is the depth-first exploration of the design space. Backtracking is supported by explicit storage of context information at several checkpoints, and by keeping backup copies of all modified files. No indication is given on the cost of keeping such redundant storage, but it is probably expensive. Nor is it clear if backtracking can be reversed, and if so under which conditions. A similar scheme has been proposed by the Task Manager [15].

Bretshneider et.al. [6] envision a knowledgeable DMS which is capable of reacting to errors discovered in the design process (e.g. a chip is too slow). The system uses its knowledge base to suggest ways to correct the design flow, or to *backtrack* to the point where the decision leading to the error was made so that the designers can revise the decision. No proof of success is provided.

In VOV, the exploration of new design alternatives corresponds to an expansion of the design trace: new transitions are added and new places are created. VOV detects conflicts if, during the new exploration, data corresponding to the old alternative is about to be corrupted. But in general the new exploration does not (or should not) destroy the old one. Under these conditions, a designer can “go back” to an old alternative by simply switching his attention to the old data and to the old parts of the trace, which have always been available. Nothing prevents a designer from pursuing simultaneously many alternatives.

Since many explorations are going to fail, the design trace will grow many branches representing these failed experiments. Although the system has no particular need to prune those branches, designers often do. VOV offers them the choice to *kill* or *forget* parts of the trace that are deemed useless. Killing some nodes means that their status becomes DEAD. This has the advantage that the killed nodes remain in the trace, as documentation of the failed exploration, but the server

ignores those nodes for most operations. Forgetting implies the deletion of nodes from the trace, as if the corresponding transitions and places had never existed in the design.

3.3 Tools that run in place

Special attention must be given to the tools that **run in place**, those tools that modify some of their inputs. These tools stress the trace model and force a distinction between information and its container.

Assume for a moment that a place in the trace represents a container of information, for example a UNIX file. A tool that modifies such place would have to declare it as both an input and an output, thus creating a cycle in the trace. But cycles are not allowed, because they violate the single assignment property, they create cyclic dependencies and they confuse the scheduling of transitions during retracing.

In VOV a place does not represent a **container** of information, but the **information** itself. A complication arises because the name used to identify a place is the name of the container, so that when different pieces of information share the same container, they are represented by distinct places with the same name.

Consider a transition t_1 that uses as input the place p stored in container $c(p)$. If t_1 produces some information that is stored back into $c(p)$ we say that t_1 runs in place. The output of t_1 is not p , which is already an input, but p' , with $c(p') = c(p)$. p and p' make a **chain** of places. If p' is used by another transition t_2 that also runs in place, the output of t_2 is p'' , $c(p'') = c(p')$, and the chain grows to include p'' . In the example illustrated by the trace in Figure 3.4, the facet `counter:logic:contents` is the container of three places: the primary input, the output of the tool `padplace`, and the output of the tool `wolfe`. (Both `padplace` and `wolfe` run in place unless otherwise specified.) The same is true for the other facet `counter:logic:interface`.

This distinction between information and container maintains the single assignment property of the trace and avoids cycles. Nevertheless, the management of tools that run in place remains complicated, because the correspondence between files and places is no longer one-to-one. The main difficulty arises because users and tools normally refer to a piece of data by the name of its container. A designer asks information about $c(p)$, rather than about p , and if $c(p)$ is the container for a chain of places it is ambiguous which place the designer means. VOV resolves the ambiguity by keeping a pointer to the *current* place in the chain. If the current place is not the one meant by the user, the user must be more specific and select the desired place in the chain by referring to it

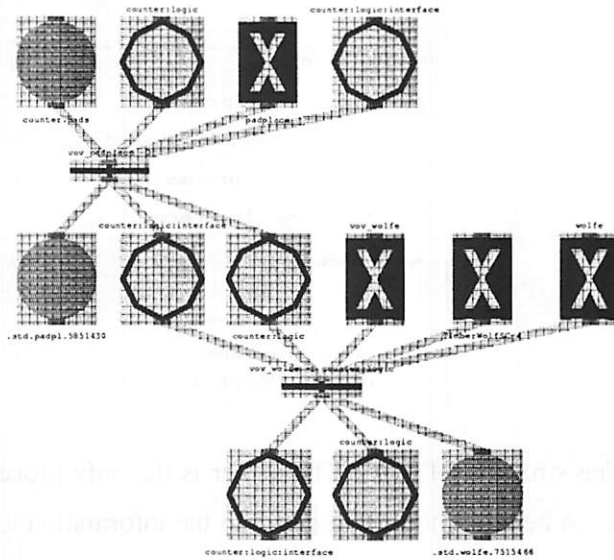


Figure 3.4: Trace left by a sequence of two tools that run in place.

as the output or the input of a particular transition.

Tools that run in place impose also some special scheduling constraints. Consider a place p that is the only input to two transitions, a regular transition t and a transition t_p that runs in place, and call p' the output of t_p . Since $c(p') = c(p)$ it is not possible to fire both t_p and t in parallel, because there may be a read-write race between the two transitions, in the sense that it is undetermined whether t would actually use as input p or p' , or worse yet, an incomplete or corrupted version of p' , as it would happen if t tries to read p just as t_p is overwriting it. A partial remedy can be provided by the database if it can prevent concurrent read-write situations, but this would still not solve the ambiguity between p and p' . VOV's solution is to schedule the in-place transitions first, on the assumption that t_p is a non-destructive transition, so that t can operate indifferently on p or on p' . If that is not the desired behavior, the designer can always modify the trace, either by making t_p not run in place, or by making t_p operate on a copy of p .

3.4 The architecture

Structure of the System

- One design, one trace, one server.
- Many tools, many designers.

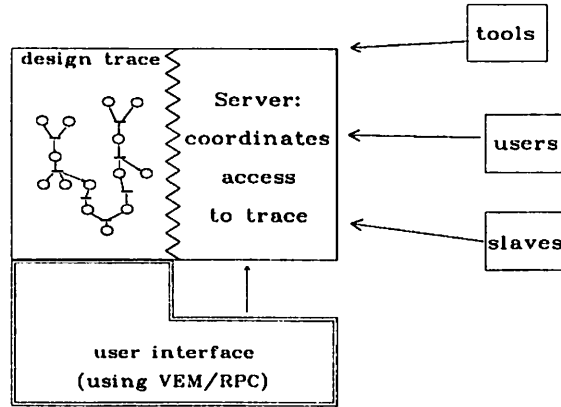


Figure 3.5: The structure of VOV. The server is the only process allowed to change the design trace. Clients can be either tools that generate the information to build the trace, users that query the server about the trace, or slaves that give the server access to CPU cycles on various machines in the local network. The graphical user interface uses VEM and RPC and it can be either a client or it can operate directly on the design trace.

As illustrated in Figure 3.5, VOV consists of several UNIX processes: a server and many clients. The **server** manages the design trace for a particular project, and is designed to run continuously for the duration of the project. The **clients** are divided into three classes:

tools are clients that provide the information to build the trace;

users are clients who allow the designers to query or modify the trace;

slaves are clients that give the server access to some resources in the network, such as CPU cycles on a machine, or access to a printer.

The slaves are the agents that execute the transitions on behalf of the server. Each slave is characterized by a list of *resources* available to the slave (e.g. large memory, printers, software licences) and by an integer number P , which expresses the *computing power* of the slave. P is defined as

$$P = \frac{K}{(l+1)t} \frac{1}{c}$$

where t is the CPU time required to execute a test routine, l is the load on the host on which the slave is running, K a normalization constant for all slaves and $c \geq 1$ is a correction coefficient

that can be specified for each slave and that is normally 1. If the load on the host is greater than a user-specified threshold, the slave refuses to accept jobs. Since l changes over time, the slave recomputes P every two minutes and transmits it to the server.

When the server determines that a transition must be retraced, it dispatches the job to the one of the slaves taking into account both the resources offered by each slave and their relative power. It is a good idea to have many slaves, connected to the server, say 10 to 20, because the more slaves are available, the more work can be done in parallel.

The bidirectional interprocess communication between clients and server is implemented with UNIX sockets. Typical UNIX implementations limit the total number of clients simultaneously connected to the server to about 60. The server services its clients sequentially in a round-robin fashion. While waiting for requests from clients, the server does not consume any CPU cycles. If no request arrives within a time-out interval, the server performs routine tasks such as monitoring of the design data, storage of the trace, and others. The time-out period ranges from a minimum of one second in the periods of activity, to a maximum of about two hours when no designer is active.

The unit of design managed by VOV is called **project**. The definition of the scope of a project is left to the user: it could be the implementation of an ALU, the compilation of a program, or the design of an entire chip. VOV is designed to handle large projects, and there is little advantage in breaking a large project into smaller ones. Each project has its own design trace.

At the beginning of the project the trace is empty. The designers use the tools normally, as if VOV did not exist, while the tools leave the trace of their execution. An advanced use of VOV also exploits the capabilities of the *assistant* to build the design trace, as described in Section 3.11.

3.4.1 Communication between the tools and the server

Whenever a tool is invoked, it should establish a connection with the server and say: I am starting now, these are my inputs, these are my outputs, I am done now. These messages are needed by the server to build the design trace.

The communication between the tools and the server is the main technical problem in VOV. Two mechanisms are available: recompilation or encapsulation.

Recompilation is the preferred option. The source code of the tool should be modified to include the appropriate procedure calls from the "VOV library," consisting of the procedures listed in Figure 3.6, and available for both C and C++ programs.

`VOVbegin()` starts the connection with the VOV server and declares that a new transi-

```

void VOVbegin( int argc, char** argv );
void VOVinput( int type, char* name );
void VOVoutput( int type, char* name );
void VOVend( int status );

```

Figure 3.6: The procedures in the VOV C and C++ library.

tion has begun. This routine should be called as soon as possible; in general it is called after the tool has established that it is going to do something, that is after it has parsed the command line options. The arguments to `VOVbegin()` describe the command line used to invoke the transition. Each input and output of the transition is declared using `VOVinput()` or `VOVoutput()`. The `type` argument is used to distinguish between OCT facets, UNIX files, executables, or the other types of a place. The name is the unique identifier of the place (see also Section 4.1.3). `VOVend()` is used instead of `exit()` to let VOV know that the transition has terminated. The argument passed to the `VOVend()` is the exit status of the process. If the transition ends without calling `VOVend()`, the server assumes that the transition has failed. If the server does not respond during `VOVbegin()`, the library turns itself off and all successive calls return immediately without effect, so that the tools can be used whether VOV is operative or not.

Sometimes it is not possible to recompile a tool. In these cases the tool is left unchanged while a **capsule** is provided to do all the talking on behalf of the tool. Figure 3.7 shows the capsule for the UNIX utility `diff`, which is a data-invariant transition. Each capsule is a *shell-script* and consists of two parts. The first part contains tool-specific knowledge to interpret the command line and derive the list of inputs and outputs for the transition. The second part is the invocation of the program `vov_capsule` which establishes the connection with the server, declares all inputs and outputs and invokes the tool. Figure 3.8 shows a more complex example of a capsule.

Recompilation is preferred because it offers the possibility of perfect accuracy in capturing all inputs and all outputs of a transition, with minimum modification of the sources and with minimum runtime overhead. Encapsulation is somewhat more complicated because the capsule must emulate the behavior of the tool to compute its inputs and outputs, which is paid with a few seconds of runtime overhead.

Capsules for data-invariant transitions can be prepared easily, while capsules for data-sensitive transitions can be quite difficult to write, because they often require, the parsing of a file. Pathological transitions may be impossible to encapsulate. Although accuracy of the capsules is a

```
#!/bin/csh -f
# Part 1: Compute inputs and outputs.
set PROG = vov_diff
set FILE1 = $1
set FILE2 = $2
set OUT = $1.$2.diff
# Part 2: call vov_capsule.
vov_capsule -c "$PROG $argv" \
-i $FILE1 -i $FILE2 -R $OUT \
diff $FILE1 $FILE2
```

Figure 3.7: Example of a capsule: `vov_diff`. The UNIX utility `diff` does not have to be modified. The program `vov_capsule` connects to the server, exchanges information with it, and then calls `diff`. The naming convention used for the capsules is that for each tool `xyz` the corresponding capsule is called `vov_xyz`.

desirable features, it is also possible to operate with capsules that are not completely accurate, in the sense that they do not declare all the actual inputs and outputs of the transition.

3.4.2 Affinity of transitions

A complete list of the attributes of places and transitions is presented in Section 4.1.2, but it is important to explain now the **affinity** attribute, which expresses the fact that some transitions require special resources to be executed. The **resource list** is an attribute of a slave, and is used together with the affinity to properly match a transition with a slave.

As previously mentioned, it is possible to connect to the server several slaves, each running on a different machine in the network. Two slaves are **equivalent** if any transition performed on one slave would give the same results if executed on the other slave. Ideally, all slaves would be equivalent and the dispatching mechanism could be based only on the greedy strategy that the most powerful slave gets the transition with longest expected duration.

In practice, different machines offer different resources and slaves can lose equivalence for several reasons: machine architecture, hardware and software resources. Some transitions, cannot be executed interchangeably on machines with different architecture. The obvious example is the compilation of a C program, because the C compiler on a SUN produces a different output than the C compiler on a VAX. Other transitions that are unusually time-consuming or that require a large amount of memory should be executed preferably on the large machines in the network. The

```

#!/bin/csh -f
set BIN = wolfe
set PROG = vov_$BIN
if ( $#argv == 0 ) then
    $BIN          # Let wolfe print the the usage message.
    exit -1       #
endif

set IN = $argv[$#argv] # Get the input facet.
set OUT = $IN          # By default wolfe runs in place.
set ARGS = ( $argv )
set ARGS[$#argv] = ""  # Clear last element in ARGS.
set OPTIONS = ($ARGS)
set INLIST = ()
set OUTLIST = ()
while ( $#OPTIONS )    # Parse the command line options.
    switch ( $OPTIONS[1] )
    case -o:
        set OUT = $OPTIONS[2]
        set OUTLIST = ($OUTLIST -o $OUT)
        shift OPTIONS
        breaksw
    case -t:
        set INLIST = ($INLIST -i $OPTIONS[2] )
        shift OPTIONS
    default:
        endsw
    shift OPTIONS
end
if ( $#OUTLIST == 0 ) then
    set INLIST = ( $INLIST -a $IN ) # Wolfe runs in place.
else
    set INLIST = ( $INLIST -i $IN ) # Not in place.
endif

vov_capsule -c "$PROG $argv" \
    -x $BIN -x $PROG -x TimberWolfSC-4 \
    $INLIST $OUTLIST \
    -A "" $BIN $argv

```

Figure 3.8: The capsule for the tool *wolfe*. Most *Octtools* have been encapsulated with a similar script.

licensing policies of some commercial programs might impose further constraints on the execution of transitions that use such programs. For example, the logic simulator for `verilog` can run only on the machines that have been licensed.

One solution to manage non-equivalent slaves could be to dismiss the problem by using the largest subset of equivalent slaves. This is unsatisfactory, either because it might be impossible to find a set of equivalent slaves that completely cover the set of resources needed to complete a design, or because such a subset could be too small and not allow an adequate exploitation of the parallelism expressed in the trace. Remember that the more slaves, the more transitions can be executed in parallel.

VOV deals with the problem of non-equivalent slaves by considering the **affinity** of each transition, as a way to express the resource *demand* for that transition. Each slave, in turn, *offers* a set of **resources**, and a transition can be executed on a slave only if there is a *match* between the affinity and the slave resource list. When dispatching a transition to one of the slaves, VOV scans the slaves in decreasing order of power, and chooses the first idle slave whose resource list matches the affinity list of the transition.

Affinity and resource lists are represented by a string of words separated by spaces. For example, if a transition can only run on a Cray, its affinity would be "cray", while the affinity of a transition that can only run on the machines called "calvin" and "hobbes," would be described by two words, "calvin hobbes". For a transition that can run on any machine, the affinity is an empty string. In a similar fashion, each slave has associated a list of resources, which defaults to a list containing the machine name and the machine type. Thus, the resource list of a slave running on the VAX "calvin" would be "calvin vax". Both the affinity list of a transition and the resource list of a slave can be controlled by the user.

A *match* between an affinity list and a resource list exists if either string is empty or if the two lists have one word in common.

3.5 Interactive tools

CAD systems are moving more and more towards automatic synthesis, but this does not mean that one can dismiss and ignore interactive tools. Some tools such as layout editors, some simulators, some data browsers, and some synthesis systems require user input. Other tools, for example simulations with graphical animations, require no input but they nevertheless require user supervision. From a DMS's point of view, batch tools are preferable to interactive tools, for the

simple reason that they can be retraced automatically at any time, while interactive tools cannot, because they require the special resource “designer.”

From VOV’s point of view, many interactive transitions are not worth tracing to begin with. These transitions are all the editing of primary inputs, which cannot be retraced, and the viewing of data, which need not be retraced. The other interactive transitions are given the affinity “interactive”, so that they can be retraced only when the designers connects to the server a special slave that provides the resource “interactive”.

An important interactive tool is called the `vov_inspector` and it allows a designer to inspect a piece of data and assume responsibility for its conformance to some standards of quality. For example, a designer might use the inspector to state that a piece of layout “looks good,” or the output of a simulation appears to be correct.

The whole design activity managed by VOV can be seen as a single interactive transition involving many tools invoked from a UNIX shell by many designers, over a long period of time. This suggests the possibility of scaling down VOV and using it to keep a trace of the activity within other shells, for example `misII`, `blis` or `slip`. The main hurdle is that most of the transformations available from these specialized shells run in-place, through destructive modifications of the state maintained in virtual memory. The trace would be a long chain of in-place transitions, without any parallelism and without possibility of a partial retracing, the only option being that of repeating the whole sequence of commands.

3.6 The firing rule

The complete **firing rule** in VOV is as follows: a transition can be forcefully fired by a designer at anytime, but during automatic retracing a transition is fired only if all the following conditions are true:

- A designer has directly or indirectly requested its firing; this condition gives some control to the designers.
- All inputs are **VALID**.
- All outputs are either **NOT VALID** or **MISSING**.
- There is an idle slave with matching affinity.

Just as important is the **success criterion**: a transition completes successfully if the following four conditions are true:

- The transition terminates normally, i.e. not because it has caught a signal.
- The exit status is one of the expected legal ones.
- All the inputs are **VALID** and their timestamp precedes the starting of the transition.
- All the outputs have a timestamp that has been modified during the duration of the transition.

If any of the conditions is not satisfied, the transition has failed. By rephrasing the success criterion we obtain a description of the four failure modes considered by VOV:

Aborted failure: occurs when a transition is aborted, for example because of a signal not caught, e.g. with `ctrl-C` or a `kill -9` command or a segmentation fault, or an arithmetic error.

Wrong exit status failure: applies to those transitions that return an unexpected status. For example, a logic simulator like `musa` normally returns zero, while other exit values mean that something in the simulation has gone wrong.

Invalid input failure: applies to the transitions whose inputs, at the time of termination, are not all **VALID**. The invalidation could have happened at any time antecedent the termination of the transition, even during execution. Transitions fired by VOV can incur in this failure only because the inputs become **NOT VALID** after the transition has been dispatched.

Missing output failure: occurs when one or more outputs declared by the transition are not found or have a timestamp that precedes the firing of the transition, this being an indication that the transition did not touch those places.

3.7 Trace versus Petri net

Like a Petri net, the trace is a bipartite directed graph, and the nodes are called places and transitions. But a Petri net allows cycles, while a trace is acyclic, more like a data-flow graph. The most significant difference is in the firing rule: in a Petri net the firing of a transition consumes the tokens in the input places, while in the trace the input places are not affected.

In a Petri net the execution of the transitions is nondeterministic. If several transitions are ready to fire the choice of which one fires first is either random or determined by external rules

that are not modeled by the net. Furthermore, firing one transition may disable other transitions that could have fired instead. The trace is deterministic because all the enabled transitions are always executed. Although the order of execution is not fully deterministic, because it depends on the availability of slaves, the final result is.

A cyclic Petri net has more modeling power than a trace. For example, the notion of exclusive use of a resource can be captured by a marked Petri net with cycles, but not by the trace. This notion appears in design management when there are transitions that require a particular resource, such as some special hardware, which can only be used by one agent at a time.

The exclusive access to a resource can be provided by an appropriate *scheduling* mechanism. A Petri net controls the scheduling with a place that represents the resource and by allowing at most one token in that place. The place is an input to all the transitions that compete for that resource. Since tokens are consumed when a transition is fired, whenever one of the competing transitions fires, all the others become disabled. When the transition completes a new token is put in the place to indicate that the resource is again available.

The trace alone is not sufficient to represent this type of scheduling, although VOV can control competing transitions using the affinity mechanism. The condition to fire a transition requires the existence of a slave that matches the affinity of the transition. For example, suppose that several transitions require a plotter, and suppose that it is not appropriate to fire more than one of those transitions at the same time. The desired behavior can be obtained by setting the affinity of the transitions to "plotter" and by providing only one slave with the resource "plotter".

3.8 Sets of nodes

There are many reasons to group nodes into a single set. A set can be used to identify all the alternative implementations of an ALU, or to group together all the steps required to route a macro-cell chip. Sets can also be used to identify the various alternatives explored in the evolution of the design, or to group versions that are part of a coordinated change to the design.

Each set is identified by its *name*. For its internal operation, VOV maintains a few sets that are characterized by a name with a leading and a trailing sequence of three semicolons:

;;;NODES;;; is the set of all nodes in the trace. This is the "universe" set in each trace.

;;;PLACES;;; is the set of all places;

;;;TRANSITIONS;;; is the set of all transitions;

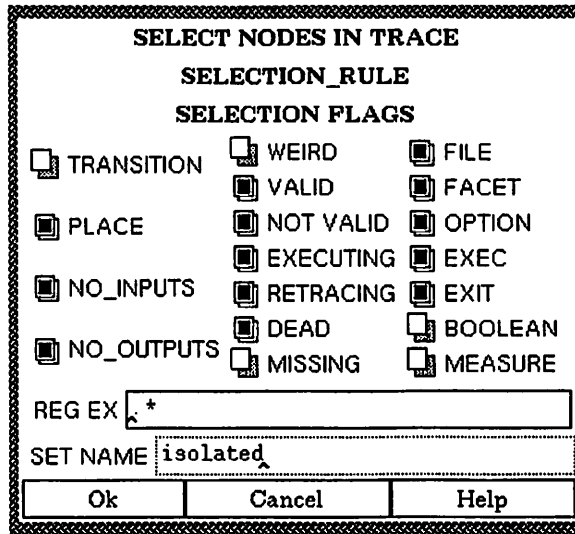


Figure 3.9: This dialog allows the user to define a selection rule for a set. The rule selects only places that have no inputs and no outputs, with any name (the regular expression “.” matches every string). MISSING nodes, boolean places or measures are not selected. The selected nodes constitute a set named “isolated.”

;;PLACES TO CHECK;; is the set of all places whose timestamp is checked at every server timeout;

;;TRANSITIONS TO FIRE;; contains all the transitions that are scheduled to be fired;

Other sets collect the nodes that have been created by the same designer, thus establishing a weak form of ownership over those nodes.

A designer can create a set by specifying the *selection rule* (Figure 3.9), which can be a rather complicated expression that considers the connectivity of the trace, the types of nodes and places, and the matching of the name of places to a regular expression.

The operations implemented on sets are complementation within another set, and union and intersection of two sets. Other operations are *fill*, *cover* and *collapse*. Given a set S and a set T , $fill(S, T)$ returns another set F containing S , T and all paths between a node in S and a node in T . The *cover* operation applies an operator to all the nodes in the set, visiting the input nodes first. The *collapse* operation is described in the next section.

3.8.1 Hierarchy in the trace

The trace supports the operation of `collapse` of a set of nodes into a single node. With this operation it is possible to model the design activity hierarchically, as shown in Figure 3.10. The inverse operation is the **expansion** of a node.

In a **simple collapse** of a set, the entire set is reduce to a single node, but not all collapses are simple, nor all sets are collapsible. A set S is collapsible only if it is *locally convex*, and it is simply collapsible if, in addition, its *boundaries are homogeneous*. A set S is locally convex if for any two nodes $a, b \in S$ with b depending on a , $b \in D(a)$, all the directed paths between a and b are contained in S : $\forall P(a, b) \subseteq S$. For example, a set with two unconnected nodes is locally convex, and so is any trace as a whole. The disjoint union of locally convex subsets might not be locally convex. Given any set A , the set $S = \text{fill}(A, A)$ is locally convex, so the `fill` operation can be used to derive a collapsible set from any set.

The second property concerns the type of the nodes in the boundary of the set. A node $n \in S$ is on the boundary $B(S)$ of S if at least one of its inputs and outputs does not belong in S :

$$n \in S, n \in B(S) \Leftrightarrow \exists m \in I(n) \cup O(n), m \notin S$$

The boundary is said homogeneous if all nodes in it are of the same type, that is they are all places or all transitions. If all the nodes in the boundary of a locally convex set are places, the set can be collapsed into a place; if all boundary nodes are transitions, the set can be collapsed into a transition.

If the boundaries are not homogeneous, the set can be collapsed into either a place or a transition, while the boundary nodes of the opposite type must be squeezed out of the collapsed node to maintain the bipartiteness of the trace. This is illustrated by an example in Figure 3.11.

Both types of collapse, to a place and to a transition, make sense from an abstract point of view; however, the collapse to a place is not used, because lumping a trace into a single place disrupts the relationship between places and database objects, and because the collapsed place might become the output of more than one transition, in violation of the single assignment property. Instead, the transition resulting from a collapse of a set has a natural interpretation as the set of operations that produce the outputs starting from the inputs, while the places collapsed into the transition represent temporary and possibly non interesting data.

Collapse is essentially an abstraction operator, for it replaces a single node where before there was a larger number, while retaining the same “interface.” The utility of this abstraction operation is clear especially from the user’s point of view, because it reduces the complexity of the

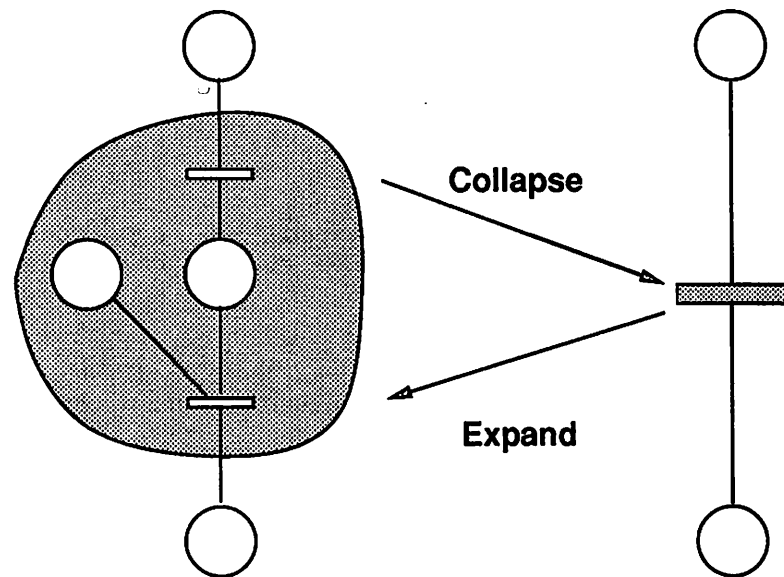


Figure 3.10: The bipartite graph allows the abstraction of subsets of the graph into single nodes. In this example, three transitions and their intermediate nodes, are collapsed into one transition.

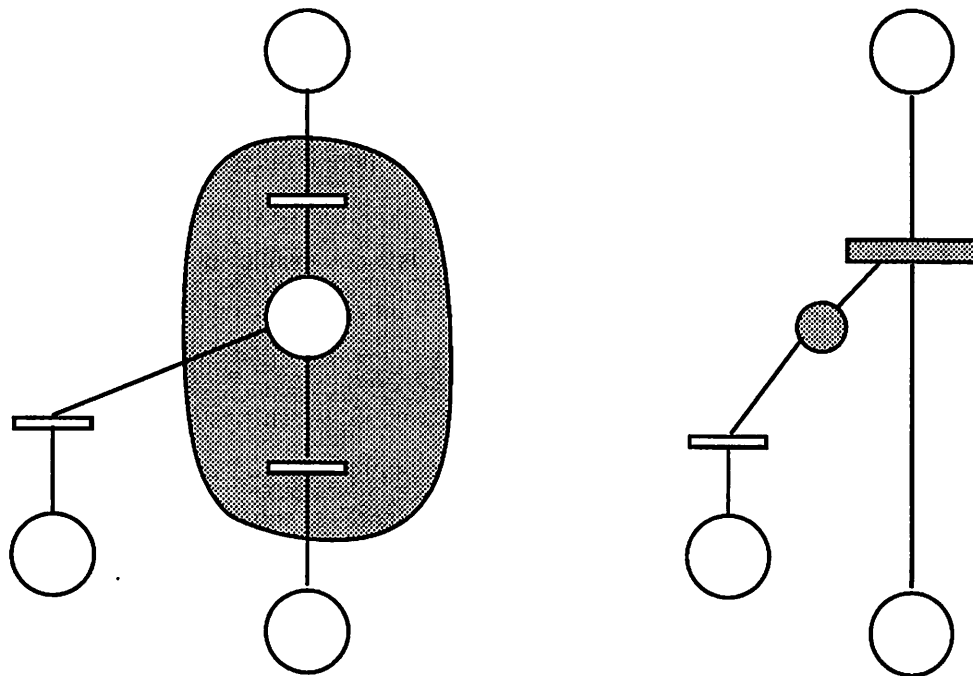


Figure 3.11: A non simple collapse due to non homogeneous boundaries of the set to be collapsed.

```
vov_mosaico -o routed chip:placed          # Route chip:placed
```

is equivalent to

```
vov_atlas chip:placed chip:placed.cd      # Channel definition
vov_cds   chip:placed.cd chip:placed.gr    # Global routing
vov_cprep chip:placed.gr chip:placed.h     # Detailed routing
vov_octflatten chip:placed.h chip:placed.flat # Flattening
vov_mizer chip:placed.flat chip:placed.mizer # Via minimization
vov_sparcs chip:placed.mizer chip:routed   # Compaction
```

Figure 3.12: Sometimes, a single command line is equivalent to a whole set of transitions. In this case, the *Mosaico* script executes all the steps necessary to route a macro-cell chip. (The command lines have been simplified.)

trace, allowing the user to understand better the flow of the design. The user interface should make extensive use of this operation. On the other hand, there is no operative advantage in having the system deal with a hierarchical representation of the trace. By lumping a part of a trace into a single transition one might lose parallelism.

A collapsible set S is equivalent to a single transition, call it T . All transitions should have at least a command line and a working directory; but what are these in the case of T ? In general, for an arbitrary S , the corresponding T is purely an abstraction, something that cannot be executed. But let's consider the reverse reasoning. Suppose that there is in fact a single command, say C , that initiates a sequence of several transitions, $\{t_1, t_2, \dots, t_s\}$. Let's consider the set S consisting of all those transitions and their intermediate places. S is a collapsible set, and its collapsed transition T can be associated with a command line, which is precisely C , and the working directory of T is also the working directory in which C was issued.

A concrete example can be found in *Mosaico*, the subsystem for placement and routing of macro-cells in the *Octtools*. The command line $C = \text{vov_mosaico -o routed chip:placed}$ starts a shell script that executes the *Mosaico* sequence, as shown in Figure 3.12. The execution of C leaves a trace consisting of six transitions, each using as input the output of the previous one. If these transitions and their five intermediate places are collapsed, the resulting transition T has *chip:placed* as its input and *chip:routed* as its output, and the command line associated with T is precisely C .

From a user's point of view it is more convenient to think of *Mosaico* as a single tran-

sition, while from VOV's point of view it is best to consider the atomic components of `Mosaico`, with the detailed description of which tool precisely produced which data. This detailed bookkeeping allows the system to selectively retrace only the components that need it, rather than the entire `Mosaico` set.

In the particular case of `Mosaico` there is no substantial difference between retracing the top level script or its components, because, no matter what the input chip looks like, `Mosaico` always expands to the same set of transitions. More interesting are the *data-dependent scripts*, i.e. scripts that may expand into a different set of transitions depending on the data. Examples of data-dependent scripts are some optimization scripts that iterate a series of steps until some criterion has been satisfied. If the input data of a data-dependent script change, retracing the components of the script may be the wrong thing to do, while the script itself should be retraced instead.

In conclusion, the collapse operation has two functions: one is to provide an abstraction mechanism to hide details from the trace, which is mostly beneficial in the user interface; the other is to create a correspondence between a data-dependent script and a set of transitions, to allow the retracing of the script itself rather than its components. Collapse is normally done "on the fly," because VOV stores the trace in its most detailed form, that is at the level of the atomic transitions, both for documentation and detailed bookkeeping purposes.

3.9 Services

In this section, we analyze the services that can be provided by a trace based system. All services use the design trace and many rely on the server/client architecture of the system.

3.9.1 Service: Design documentation

The design trace, which is automatically and non-intrusively recorded, describes the detailed **history** of the design. It is, of course, a special notion of history, the one represented by the trace, because it can be rewritten and edited, and some parts can be forgotten. For example, if file `f1` was once declared as the output of transition t_1 and later redeclared as the output of transition t_2 , the trace would only have a record of the most recent declaration, the older one being forever lost. Even without being a pedantic historical record, the trace can answer important questions about the current status of the design: How was this place obtained? What data depends on this place? Did anybody run the compactor on this chip? How long did it take to obtain this place?

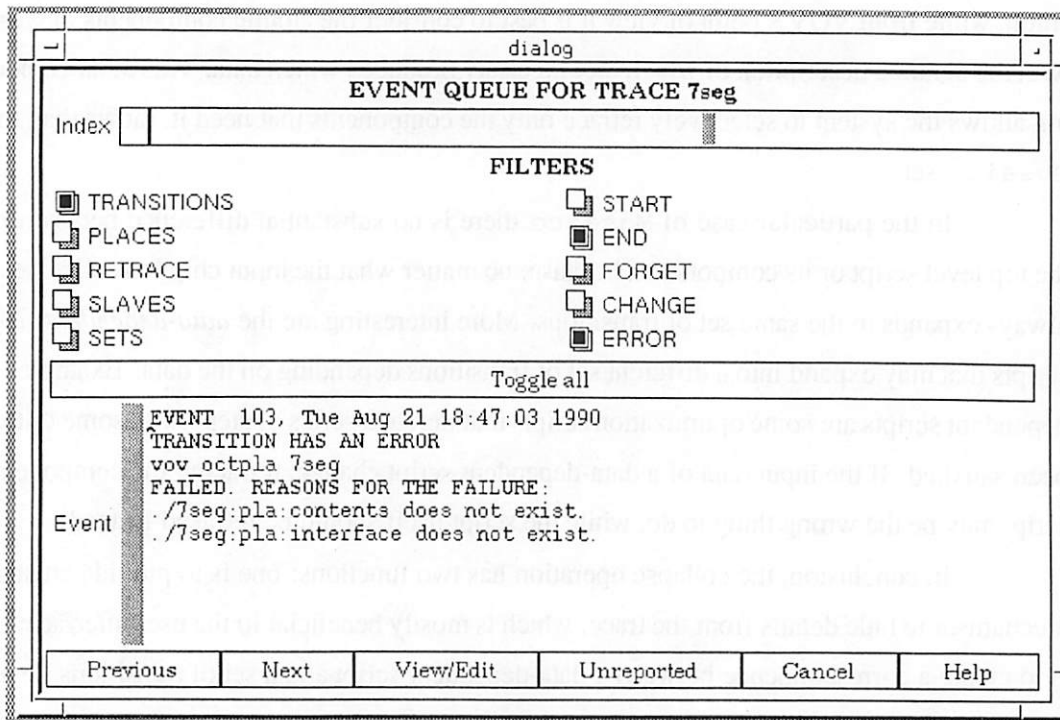


Figure 3.13: The dialog to browse the event queue allows the designers browse only the events they are interested in.

The trace is a factual record of *what* happened in the design. A complementary **annotation mechanism** allows the designers to attach a note containing some text to any object in the trace, for example to describe *why* some transition has been executed, or the *meaning* of a place, or why a set of nodes is important.

Any creation, modification, deletion, initiation, termination of a place, transition or set is an *event*. All events are permanently recorded in chronological order in an ASCII file called the **journal**. Recent events are also stored in an event queue for easy inspection by the designers (Figure 3.13). The journal is normally used only to analyze old events, a rare occurrence according to experience.

The designers can ask many questions about the status of the design. The server maintains data on the size of the trace, a list of the last few transitions initiated by each designer, a list of all the transaction currently executing, and the status of all the slaves connected to the server. Figures 3.14 and 3.15 show some typical responses from the server.

```

VOV_SH: DEVELOPMENT VERSION: 2.05: FSM@eros
TRACE: PAGE FSM.page.0
      PLACES: 35  TRANSITIONS: 8  SETS: 0
USER paul IDLE: 10.4 hours
      vov_nova fsm.nova
      vov_cp fsm.nova.esp fsm.esp
      vov_mis_esp2blif fsm.blif
      vov_octpla fsm
USER <vov_server> IDLE: 20.5 hours
SLAVE 1:      eros: 160: IDLE
SLAVE 2:      fjord: 144: IDLE
SLAVE 3:      orca: 146: IDLE
SLAVE 4:      papaya: 148: IDLE
SLAVE 5:      peking: 151: IDLE

```

Figure 3.14: The server is queried about the current status of the design. User Paul has been idle for more than 10 hours. There are 5 slaves connected to the server. All of them are idle, and the most powerful, with a relative power of 160, is the one running on eros.

```

$ vov_sh -h ccdp:padpMG

/net/canova/users/casotto/yu/ccdp:padpMG:contents
VALID Sat Nov 3 12:33:28 1990 (canova)

VALID vov_padplace -g ccdp:symbolic
VALID vov_puppy -o fl0:puppy fl0:symbolic
VALID vov_makeSoft ccdp:symbolic
VALID vov_makeSoft mmdpcntl:symbolic
VALID vov_makeSoft pcreload:symbolic

```

Figure 3.15: The history of the object called `ccdp:padpMG`, as reported by the server. The place is VALID, and it is the immediate output of the first transition in the list. The other transitions are those in the path from the mentioned place backwards to the primary inputs of the trace.

3.9.2 Service: Data monitoring

When the server times-out for lack of client requests, it performs some light-weight routine tasks. One of these tasks consists of checking the time-stamp of some places in the trace. Young places are checked every time, while places that are more than three days old are checked only about once an hour, the rationale being that places with a recent timestamp are more likely to change than places that have not been touched in a long time.

This continuous monitoring is computationally cheap, and it is actually not essential for the correct behavior of the system, because time-stamp checking must be performed again in order to process correctly many client requests. One reason to perform this monitoring is to get a detailed chronology of events, another is that some tools might fail to declare some of their outputs, and this paranoid checking is the best hope to detect those faulty tools.

If a change in a VALID place is detected, for example because the place is a primary input and a designer has just finished editing it, all the dependent nodes become out-of-date and are therefore marked as NOT VALID. A less likely event is the change of a NOT VALID place, which happens only when VOV is not notified of a design activity due to malfunction of the tools (or of VOV itself). In this case the place becomes VALID while all its dependents become NOT VALID.

3.9.3 Service: retracing

Whenever a transition becomes NOT VALID, either because one of its inputs has changed or because the transition itself has been modified, the most common action to re-establish consistency is to run the transition again. The trace contains all the information necessary to repeat any transition in the trace, namely the working directory and the command line. As mentioned before, the repetition of a transition in the trace is called **retracing**.

Retracing is always initiated by a user's request and it can be global or local. A *global retracing* schedules for retracing all the invalid transitions in the trace. *Local retracing* comes in two flavors: retracing with a *target* place or retracing with a *source* place. The first activates all transitions that are necessary to update the specified place. The second form activates all transitions that depend upon the specified place. Global retracing is most often used during small, single designer projects. Sets of places can also be used as targets or sources for local retracing.

Among all the transitions scheduled to be retraced, the server dynamically builds a list of those transitions that are immediately ready to be fired, i.e. the transitions whose inputs are VALID and whose outputs are not. The server dispatches each ready transition to one of the slaves, starting

from the transition with the longest expected duration, which is the time taken by the transition the last time it was executed. Each transition is dispatched to the idle slave with the maximum relative power, among those slaves whose resource list matches the affinity of the transition. If no match is found, an error event is generated, because there is no slave with the resources required to execute the transition, and the transition is removed from the retracing set. Dispatching is repeated until all slaves are busy or until all the ready transitions have been dispatched. Upon termination of a retracing transition VOV repeats the entire dispatching algorithm, both because a slave has become available for another transition and because new transitions may have become ready to fire.

This dispatching mechanism is useful to exploit the power of those machines that would otherwise be idle, because if a slave is running on an unloaded machine, its relative power is large compared to slaves running on similar loaded machines.

The retracing mechanism allows VOV to emulate the UNIX utility `make`, with the difference that no `Makefile` has to be prepared, and that retracing can use multiple slaves to exploit the parallelism in the trace.

Retracing detail

Retracing is complicated by the fact that the input/output dependencies of a transition are recomputed at runtime, and in some cases the dependencies of the retracing transition may be different from the dependencies already recorded by the trace. This happens, for example, with data-sensitive or pathological transitions.

Let's consider the compilation of a "C" file. Suppose that the file contains the line

```
#include <foo.h>
```

When the compiler `vo_v_cc` is run, it indicates to the server that the file `foo.h` is used as an input. Suppose now that the `#include` statement is deleted. Since the file has been modified, the compilation should be repeated. But the compilation no longer uses `foo.h` as an input, and the trace has to be modified to take this into account. This type of events is normal but relatively rare. Since it requires a change in the topology of the design trace, it is considered important enough to notify the designers by posting an event on the event queue.

VOV detects these changes in the traces due to retracing by comparing the inputs and outputs of the old and of the new transitions. In detail, this is what happens:

1. The server dispatches the transition T to a slave: the status of T is changed to RETRACING.
2. The slave uses the description of T (working directory and command line) to initiate a new transition T'.
3. The new transition T' connects to the server, just like any other transition initiated by a designer, but this time the server recognizes that T' is a retracing transition because it is identical to T; the status of T' is set to TRACING, while T remains RETRACING. Some of the attributes of T which should be preserved during retracing and all the annotations are copied from T to T'.
4. T' declares its inputs and its outputs; some outputs of T' are also outputs of T, which causes those output places to have two incoming arcs, in temporary violation of the rule that a place can have at most one input.
5. If T' aborts, it is deleted from the trace and the retracing of T has failed: T and its outputs become NOT VALID.
6. If T' terminates normally, the server compares the input/output lists of T and T'; if they differ, the designers are notified. The criteria to determine whether a transition has failed are applied to T'. If T' succeeded, T' and its outputs become VALID, otherwise they become NOT VALID.
7. The slave communicates that the retracing of T is now done and the server deletes T; old outputs of T that are not outputs of T', become NOT VALID and isolated.

Retracing deadlocks

Some sequences of events can lead to situations in which a transition cannot be retraced. Suppose that, because of a mistake or a bug, a transition declares as input a place that does not exist; the transition fails for lack of input and becomes NOT VALID. The condition that enables the retracing of the transition is that all of its inputs are VALID. But a place that does not exist cannot possibly be VALID, and the transition is therefore blocked into its NOT VALID state. This deadlock situation requires the intervention of a designer, who has two options to proceed: the offending place can be removed from the input list of the transition, thus changing the condition upon which the transition becomes ready to fire, or the correct transition can be manually executed.

Another pathological situation can be caused by transitions that do not describe their input/output behavior accurately. Suppose that a transition modifies one of its transitive inputs, but does not declare the place as its output: an example is a tool that runs in place without informing the server. All dependents of undeclared output, and the offending transition in particular, become invalid and have to be repeated, only to cause the same events to repeat, and so on in an infinite loop. Since retracing must be explicitly requested by the designers, it can be argued that a loop that includes human intervention is not really an infinite loop. In any case, the information produced by the system, the journal in particular, should be enough to track down the reason for the abnormal behavior.

3.9.4 Service: Conflict detection

The server can detect four types of conflicts:

Input conflict: An input conflict arises whenever a transition uses an input that is not VALID. The designer who has invoked that transition is notified of the problem and given a chance to avoid it, for example by aborting the transition. However, it often makes sense to continue the transition anyway, for the purpose of establishing a path in the trace, and let VOV repeat the transition whenever its inputs become VALID. If the transition had been invoked by VOV, the transition is always aborted.

Output conflict: An output conflict occurs when a transition declares as output a place that is already the output of another transition. But a place can be the output of at most one transition, and the designer is asked if he really wants to forget the old transition and replace it with the current one. Data-sensitive and pathological transitions can cause this conflict even in the case of transitions invoked by VOV, in which case the transition fails.

A special case of this type of output conflict applies to primary inputs. VOV notifies the user if a place which was a primary input is about to become the output of a transition, or if a tool is run in place on a primary input, because in both cases there is a risk of loss of data.

Lock conflict: Some operations, such as the editing of places, sets or transitions, require the exclusive access to some objects in the trace. This is provided by software locks managed by the server. An attempt to lock an already locked object causes a lock conflict. A lock can always be broken, but the user breaking it assumes responsibility for the action, which is also recorded in the journal.

Cycle: The trace cannot contain cycles, and the server must check each dependency declaration to make sure that it does not introduce one. Experience shows that this kind of conflict is rare.

Conflicts are detected in real time by the server. Conflict detection should be performed efficiently, because it adds an overhead to most interactions between tools and the server. In the current implementation, such overhead has been measured to be on the order of a few hundredths of a second, which is acceptable.

3.9.5 Management of refinements and alternatives

In the literature the word *version* is overloaded to mean both *refinement* and *alternative*. These two concepts should be distinguished, because they have different representations in the design trace.

A **refinement** of a place is a modification that invalidates the current status of the place. Correcting a typo in an ASCII file is a refinement, because no designer is interested keeping a copy of the file with the mistake. In a refinement, the previous “version” of the object is no longer of interest, and it is effectively lost, unless the designer uses some independent revision control mechanism, such as the UNIX RCS. The refined place remains VALID, but all the dependent places become NOT VALID, to indicate that they are no longer up-to-date with the recently refined place. A refinement is represented by a process of invalidation of nodes already in the trace, as illustrated in Figures 3.16 and 3.17. Figure 3.16 shows a fragment of a trace in which all the nodes are VALID. The trace refers to the design of a counter starting from the behavioral description `counter.bds`. Imagine that the designer realizes that the `counter.bds` has an error, because instead of describing a counter with an active-low reset signal, it describes an active-high reset. The designer has no interest in keeping around the faulty description. Using a text editor, he refines `counter.bds`. The server notices the change and invalidates all nodes depending on `counter.bds`, as in Figure 3.17, where the different color of the nodes denotes that those nodes are NOT VALID. At this point, the designer may request a local retrace from `counter.bds` to recover consistency.

A new **alternative** is represented by adding new nodes to the trace. An alternative for a design object is created whenever one wants to try something new without destroying the current status of an object. Alternatives are therefore distinguished by their history.

The trace does not capture the notion of “equivalence” between two alternatives, because such notion is context dependent and extraneous to the design management problem. For example, suppose that a designer has a standard-cell implementation of a controller and suppose that he wants

Figure 3.17: Refinement: trace after refinement. The file counter . bds in the top row has been refined, and all its dependents have become NOT VALID.

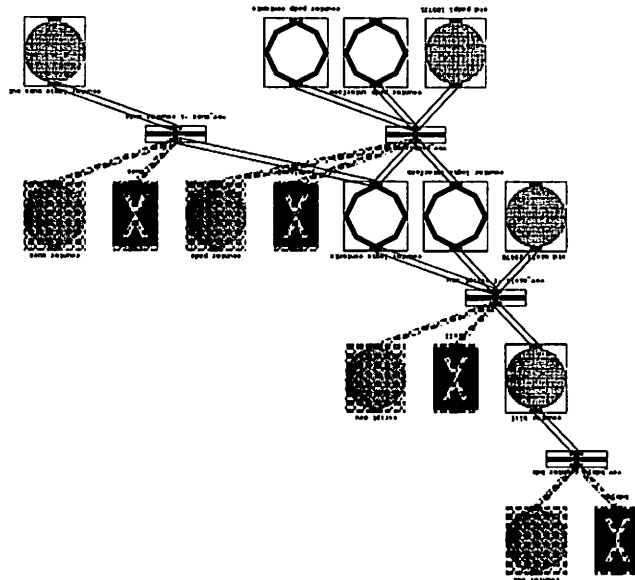
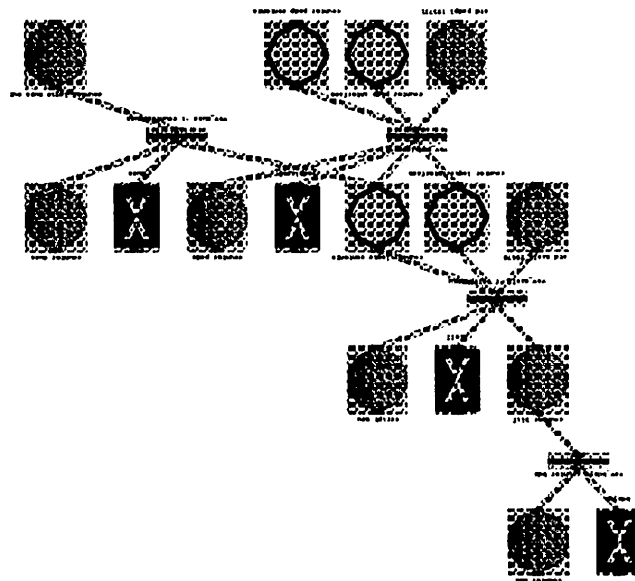


Figure 3.16: Refinement: trace before refinement. All nodes are in the VALID state.



to explore the possibility of a PLA implementation. The design methodology to build a PLA is quite different from the methodology required by a standard-cell block. The standard-cell implementation and the PLA might be equivalent in their logic behavior, but they are probably quite different from both the layout and the performance points of view.

Figure 3.18 shows a step by step growth of an alternative. In the top left we see a trace fragment that describes the synthesis of a FSM starting from the textual description contained in the file `fsm.bds`. Suppose that the designer decides to try a different state encoding. He might, for example, copy the primary input `fsm.bds` into `fsm1.bds`, and edit the new file to represent the new encoding (top-right). Regardless of the state encoding, the design methodology for the new FSM is probably going to be quite similar to the previous one. With the same techniques used by the automatic assistant (described later in Section 3.11), the designer can ask VOV to automatically extend the trace from `fsm1.bds` following a flow similar to the one used for `fsm.bds`. Depending on the case, the extension can stop at the first step (bottom-left) or continue all the way (bottom-right).

3.9.6 Archiving

One of the requirements for an ADM is to help archive a design. Assuming that all the tools leave a complete trace of all their inputs and outputs, the trace can become the primary source of information to determine which data should be saved in the archive. By distinguishing between deterministic and non-deterministic tools, it is also possible to save storage space by not archiving the outputs of all deterministic tools, given that they can be regenerated automatically.

3.10 Use of measurements

Measurements on the design data are needed to provide important services such as verification of design specification, validation of a piece of design data, and design estimation. A method to measure the area of a circuit is needed to help the designer choose the smallest layout. A method to decide whether a circuit has been successfully simulated (a “Boolean” measure) is needed to inform the designers that more simulations should be performed. A method to count the instances in a standard-cell netlist is the basis to build a predictor that estimates the area of a standard-cell implementation given the number of instances in the initial netlist (assuming that such an estimate is meaningful).

The measurement itself is just another type of place, more specifically a place which is the output of a **measuring tool**. Each measurement consists of a value and a unit of measure. The measurements can be kept up-to-date by the VOV with the normal built-in retracing mechanism. The difference between a measurement and other places is a bidirectional link that associates each measurement with the place which has been measured. These links are established at runtime by the measuring tool itself.

The area of a chip can be computed by the tool `vov_meter`, which, upon execution, declares the chip as input, its area as output and establishes the link between the chip and its area. In this way, it is possible to query about the area of the chip and know whether the reported number is up-to-date or not.

Figure 3.19 shows a trace containing measurements. Here the tool `vov_meter` is used to count instances and to measure chip area and total net length.

The knowledge required to perform the measurement is completely contained in the measuring tool. The DMS is not required to know anything about area or speed or any other measurement; only the measuring tools do.

Groups of measurement can be handled as regular sets of nodes, but new operators are defined on these sets: a *reduction* operator is defined to find the smallest, largest, sum, product of the measurements in the set. Besides the *name*, common to all sets, each measurement set is characterized by an *objective* which describes whether the goal is to simply monitor the set, or to minimize or maximize the figure of merit, or otherwise make it belong to a “good” set.

A technique that we plan to explore is to accumulate statistics on design parameters and on the corresponding performance indices. For example, considering a trace relative to the execution of `Mosaico`, the ADM can relate the total number of cells in a chip to its total area after placement and routing. This information can be converted into a lookup table and interpolation techniques can be used to obtain predictions for different values of the design parameters.

3.11 The assistant

VOV has been designed principally with concern for the needs of experienced designers; for their benefit, the system is non-intrusive and non-restrictive. The expert designer has complete freedom to choose the tools to be used in any situation, while VOV intervenes only in case of conflict, as an adviser that warns the designer about potential loss of data or potential waste of CPU cycles.

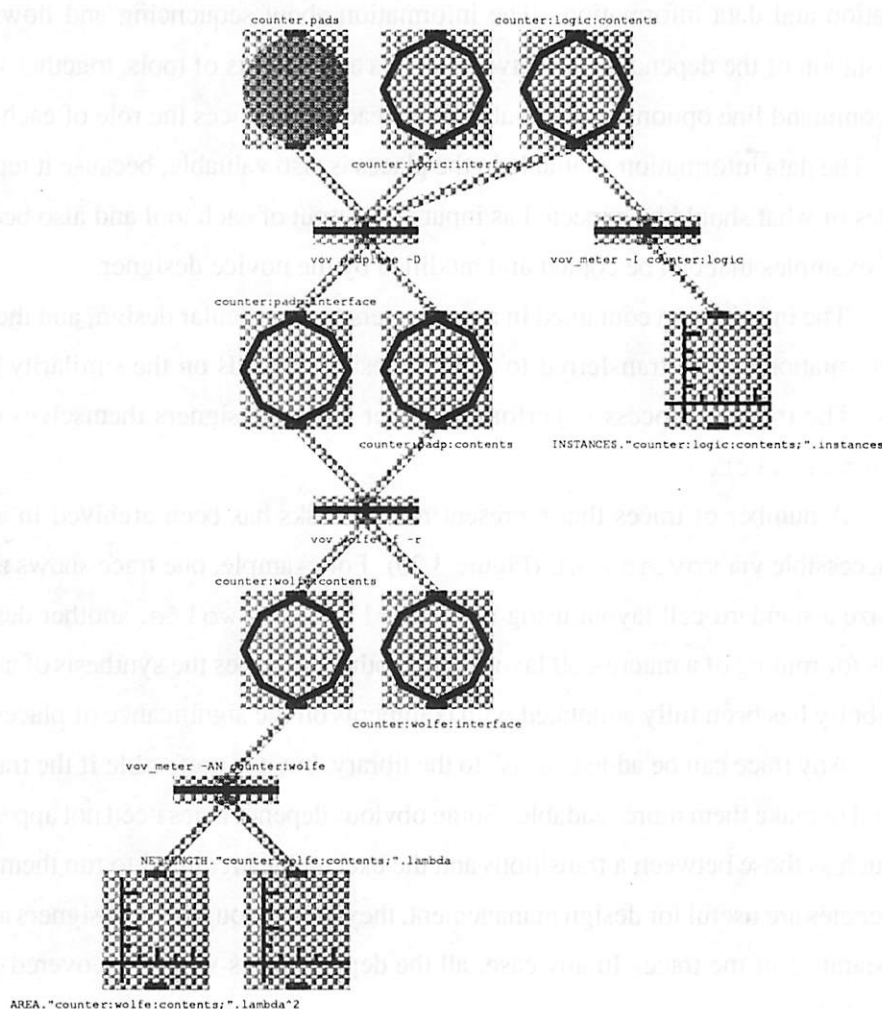


Figure 3.19: A trace complete with measurements. The tool `vov_meter` counts the number of standard cells in the input netlist to the tool `wolfe`, and then measures area and total net length of the placed and routed layout.

Not all designers are experts. A much larger segment of users of CAD systems are relatively inexperienced, or altogether novices, and they need assistance and guidance. For them, a new resource is available: the collection of traces of previous designs.

A trace contains two types of information that can be useful to a novice designer: flow information and data information. The information about sequencing and flowing of tools, the representation of the dependencies between inputs and outputs of tools, together with examples of use of command line options, are all valuable in teaching novices the role of each tool in the CAD system. The data information available in the places is also valuable, because it represents concrete examples of what should be expected as input and output of each tool and also because it provides a set of examples that can be copied and modified by the novice designer.

The information contained in a trace refers to a particular design, and the degree in which such information can be transferred to another design depends on the similarity between the two designs. The transfer process is performed either by the designers themselves or by a program called `vov_assist`.

A number of traces that represent routine tasks has been archived in a library, and are easily accessible via `vov_assist` (Figure 3.20). For example, one trace shows the normal way to synthesize a standard-cell layout using `bdsyn`, `misII`, and `wolfe`, another describes the use of the tools for routing of a macro-cell layout and another describes the synthesis of a FSM. Each trace in the library has been fully annotated with comments on the significance of places and transitions.

Any trace can be added “as is” to the library, but it is preferable if the traces in the library are edited to make them more readable. Some obvious dependencies need not appear in the example trace, such as those between a transitions and the executables required to run them. Although these dependencies are useful for design management, they are obvious to the designers and tend to clutter the appearance of the trace. In any case, all the dependencies will be recovered as the transitions are executed.

The simplest way to extract information from a trace is to let the designers do it themselves. Humans have a peculiar ability to **learn by example**; they find it easier to modify a piece of data rather than to create it from scratch. The effectiveness of examples has been shown in [12], where it is reported on the successful use of traces to teach students in a VLSI design class the proper sequencing of the *Octtools*.

A novice designer can study an example trace either graphically or textually, to understand its meaning. He can modify the data and the command lines to fit his own goals, and then he can execute the tools directly.

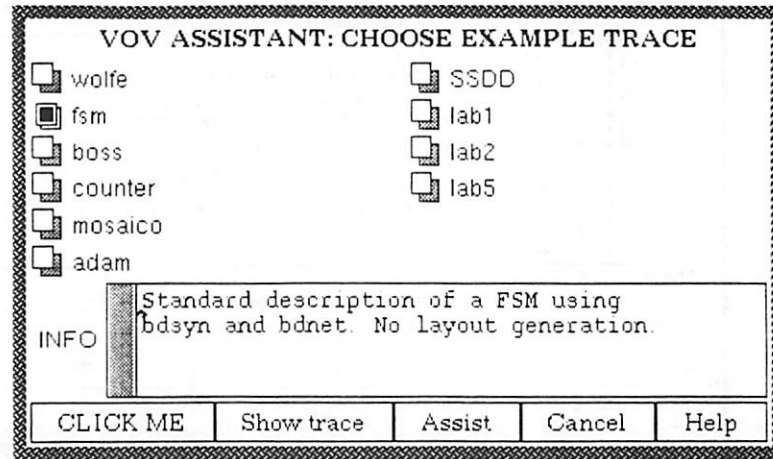


Figure 3.20: The menu to select an example trace. The trace called fsm has been selected, and the INFO field shows a short description of the trace.

The program `vov_assist` automates this process of modification of command lines. The assistant considers a trace as a particular execution of an unwritten program, it tries to deduce the program from the trace and then tries to guess the effect of running the program with different inputs. From an abstract point of view this is a difficult task, because there is the problem of recognizing conditional and iterative structures in the program, and because inputs and outputs of many transitions cannot be known until the transitions are executed. The assistant makes two simplifying assumptions:

1. A trace can be transformed into a program by simply performing **substitution of variables** in the command lines of the transactions and in the names of the places.
2. The transitions represented in the trace are not pathological and the trace captures the non data-sensitive dependencies of each transition.

For most traces one or two variable substitution rules are sufficient. The current implementation of the assistant allows up to five (Figure 3.21). Heuristics are used to suggest automatically one or two variables that may be substituted.

The **basic operation** performed by the assistant is the copy of a transition from the example trace to the current design trace. After applying variable substitutions to the command line and to the current working directory of the example transition, the assistant emulates the interaction of the transition with the server, without executing the transition. The assistant connects with the server, declares that the transition has started, declares as inputs and outputs the same inputs

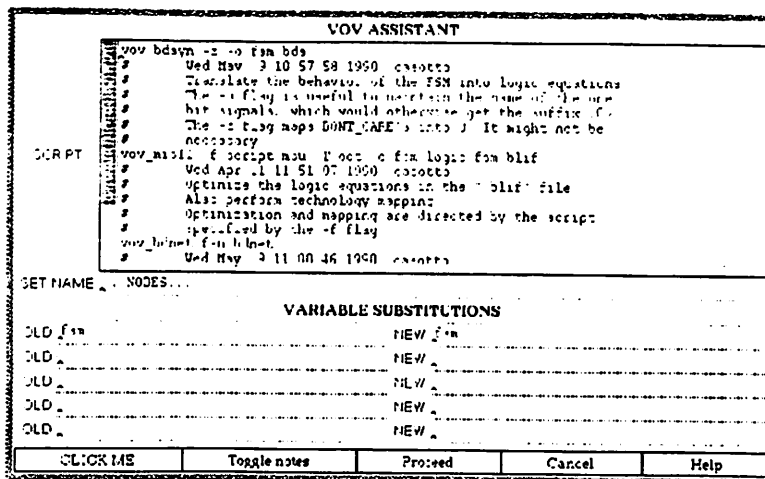


Figure 3.21: A trace has been chosen, and the designer must now specify the variable substitution rules.

and outputs of the example transition, although the names are modified by the variable substitution routine. If an input place of the transition is also a primary input of the example trace, the assistant attempts to physically copy the place into the current working directory, unless the place already exists. Then the transition is declared completed, and its status is made NOT VALID. The entire example trace can be copied in this way, one transition at a time.

The program operates in both interactive and batch mode. It is possible to restrict the assistant to operate on a subset of the example trace, and the current design trace itself can be used as example.

After the assistant has completed its job, the designer can modify the primary inputs and request a retracing. With retracing, all dependencies are recomputed, and the trace will appear just as if the designer had directly invoked all the transitions in it.

The assistant was designed originally for the benefit of novice users, but it has become a useful tool for the experts as well. For example, anyone who has ever designed with the *Octtools* has probably needed a standard-cell counter. Given that this is a routine activity, a trace called *counter* has been included into the VOV library. This trace represents the design of a 5-bit binary counter implemented with standard-cells, including simulation and layout evaluation. The behavioral description of the counter is contained in an ASCII file called *counter5.bds*. The description is parameterized, as allowed by the BDS languages, so that it is easy to describe another counter with a different number of bits.


```
#!/bin/csh -f
set BLOCK_LIST = ( b1 b2 b3 b4 )
set CHIP = mychip
foreach i ( $BLOCK_LIST )
    vov_assist -p wolfe -N $i # Describe and implement blocks.
end
vov_assist -p simul -N $CHIP # Simulate and verify.
vov_assist -p mosaico -N $CHIP # Place and route.
```

Figure 3.22: A shell script can be used to specify a complex methodology as a combination of several example traces processed by the VOV assistant.

A designer, expert or novice, who needs an eight bit standard-cell counter can invoke the assistant in batch mode, by typing for example:

```
vov_assist -p counter -N 8bitCounter
```

`vov_assist` copies the `counter` trace into the current trace, and it copies the primary inputs of the example trace in the working directory. The variable substitution mechanism replaces all occurrences of the string `counter5` in the trace with the string `8BitCounter`. The string `counter5` is deduced automatically by the heuristics coded in the assistant. The behavioral description of the counter is now in the working directory, in a file called `8bitCounter.bds`. The designer can now edit the file, change the 5 into an 8, ask for a retracing, and the 8 bit counter is done.

Sometimes only the flow is of interest, while the primary inputs of the trace have no relation with those in the example trace. Suppose one wants to build a standard-cell decoder, using a design flow similar to the one required for the counter. Suppose also that the file `decoder.bds` has already been prepared in the current working directory. The command

```
vov_assist -p counter -N decoder
```

uses the assistant to transfer the flow in the example trace called `counter` into the current design trace. The file `decoder.bds` is left untouched by the assistant, and the designer can immediately ask for a retracing.

The batch interface to the assistant allows the description of complex methodologies in a hierarchical fashion as a combination of smaller example traces, as shown in Figure 3.22, which shows a shell script that produces the complete trace for the design of a standard cell chip.

3.12 Support of design methodology

The assistant can also be used in situations where a well defined **design methodology** should be adopted by the designers. A set of traces, complete with annotations, sets and measurements, can capture the design methodology, including alternative paths. The traces can then be merged with the trace of the current design to guide the designers along the correct design methodology.

The current version of VOV does not allow strict enforcement of a methodology, on the assumption that designers work better if they have freedom. The traces created by the assistant only help guide the designers, but they can be overridden, if the designers decide to do so, by simply executing other transitions.

In some cases enforcement of a methodology can be desirable, although enforcement is usually paid with a limitation of the designers' freedom. For example, a design division in a company may have determined that it is more effective to postpone all layout activity until all the components of the chip have been described, documented, and simulated. This constraint can be represented in the trace by a transition whose output is a set of places required by the layout tools, and such that the transition succeeds only when a certain condition is satisfied, for example when high-level simulation of the chip has been completed. Such transition could correspond to an interactive tool that succeeds only when an authorized designer, for example the project manager, says that it should succeed and takes responsibility for this decision. Automatic tools to perform such a function are also conceivable. The bottom line is that the description of a methodology requires the inclusion in the design flow of special tools that contribute the knowledge specific to the methodology.

The actual enforcement of a methodology is achieved by restricting the power to modify the topology of the trace to some special clients, namely the assistant and the tools invoked by VOV during retracing, thus preventing the designers from changing the design flow by executing other transitions. Although this enforcement technique may not be bullet proof, it is overshadowed by a more challenging enforcement problem: to make the designers use VOV throughout the design. But this problem is beyond the scope of this research, because it belongs in the area of discipline enforcement within a design group.

Some *methodology traces* can be created by simply going through an example design. The resulting traces can then be installed in the library, where they become available to all designers. Other methodologies can be described as combination of example traces as already mentioned in

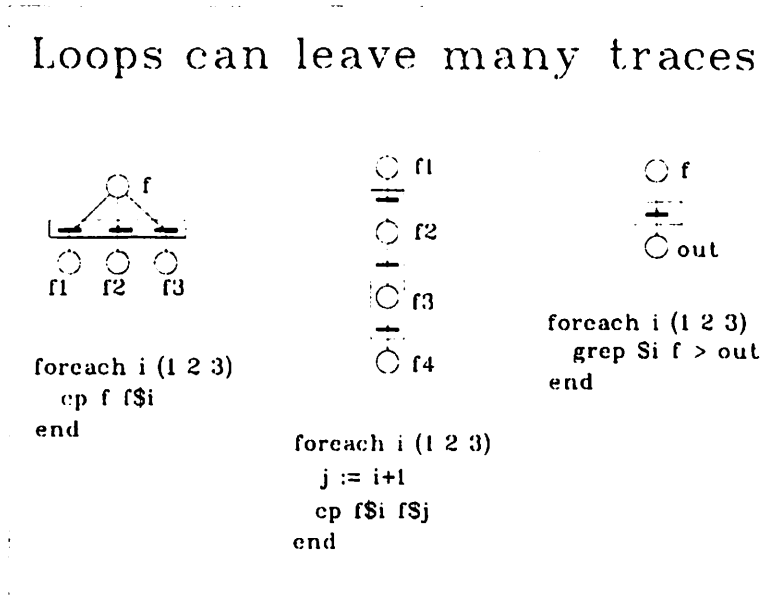


Figure 3.23: The same control structure can originate different traces depending on the operations performed in the body of the loop. These examples use simple UNIX shell scripts.

the previous section and shown in Figure 3.22.

3.13 Iteration in design

In principle, all iterations can be reduced to the following elements [1]: a **loop initialization**, a **loop body**, a **stopping criterion**, a **next-value function**. Nevertheless, the execution of each loop can leave its own peculiar trace, depending upon the actions performed in the loop body. Figure 3.23 shows three loops, all having the same control structure. In the first loop, each iteration is independent of all the others and the corresponding trace consists of a collection of parallel branches that can be executed in parallel or in any order. In the second loop each iteration uses results from the previous one, thus establishing a precise ordering among the iterations. In the third loop, each iteration invalidates the previous one, so that only the last iteration is useful.

In some sense, the design activity as a whole is a loop, whose stopping criterion is the satisfaction of the design goals, and the body consists of whatever the designers decide to do. Although this might sound trivial, the main form of iteration in design is expressed by the following tautology: *while the design is not finished, continue the design*. These iterations take the form of successive refinements of the design data and of the set of dependencies among them. This basic

design iteration is supported by VOV's tracing and retracing mechanism. In the rest of the section, other types of iteration are considered, such as optimization loops, and data-dependent loops.

Systems like ULYSSES [11] and Cadweld [17] provide limited support for iterative design, by means of the backtracking mechanism (see Section 3.2.1). The Siemens system [6] supports a specific form of iteration which can be called "goal oriented refinement," in which a sequence of operations is iteratively applied until some goal has been achieved, for example until the timing of the chip is correct, or until a satisfactory fault coverage has been reached. The designer is directly involved in the loop because he must decide when to terminate it, while the ADM supports this decision making process by consulting its rule-based expertise. Another form of iteration is also supported, which is based on the hierarchical structure of a design, in the sense that it allows the specification of conditions that should be satisfied by all the subcomponents of a module. The Task Manager [15] considers iterative processes such as tight edit/compile loops, although the loop is completely controlled by the designer. The iterative nature of the process must be declared explicitly by the designer at the beginning of the loop, and it must also be terminated explicitly. Such declarations are necessary to suspend temporarily the automatic backup mechanism for the design data involved in the loop.

For VOV, the least-effort approach was chosen. Rather than supplying its own control mechanisms, VOV tries to promote the use of existing mechanisms, in particular the UNIX shell scripts.

One technique is to **hide** the iteration from the design manager. This can be done by writing a shell script that contains all the elements of a loop, the initialization, the body, the next value function, and by considering such script as an atomic transition. In this way, the iterative nature of the task performed by the script is completely invisible to VOV. These scripts can be written on demand and effortlessly incorporated in the design flow as new tools. This is possible because a trace based system is open to the introduction of new tools, due to the great flexibility derived from the absence of an a-priori description of the tools that can be used in a design.

The hiding technique is general, effective, and free, because it uses existing and familiar software. However, in many ways, it defeats the purpose of this research, because it subtracts a part of the design activity from the control of the DMS. This may be totally acceptable when the scripts are self-contained and call fast and reliable tools, otherwise it may be better to try to use the services of the DMS, to avoid conflicts and wasteful repetition of expensive transitions.

The second technique uses the mechanism of collapsable sets. Once again, a script is used to represent the control logic of the loop, with the difference that the script is not executed as

an atomic transition, but as a set of many transitions, each leaving its trace. This corresponds to unfolding the iteration, although nothing can be said about the structure of the trace upon completion of the script, as reminded by Figure 3.23. All the transitions invoked by the script are collected into a set, call it S . The set S , or more precisely $\text{fill}(S, S)$, is collapsable, and the collapsed transition corresponds to the invocation of the script. This technique offers all management services during the execution of the loop, it allows local retracing of parts of the loop, and it also allows the retracing of the entire script.

This second technique is insensitive to the type of iteration described in the script. It can be a data-dependent loop, an optimization loop, a search loop, or even not a loop at all. Through the command interface to VOV (section 4.3.2), the scripts can also request services, such as retracing, or query the server for information, for example about some measurements or about the status of a place.

A third technique to support iteration is offered: the designer can define a set of nodes and then perform an operation *for each* node in the set. This is a more restricted technique, because only a limited set of operations are allowed. These include many operations on nodes (edit, forget) and on node attributes (status, name, affinity, etc.) most of which are performed directly by the server.

3.14 Principles that guided the development of VOV.

The development of VOV has been characterized by the application of some ideas that worked and by the exploration of others that did not. In hindsight, it is important to become aware of those principles which have been most successful: simplicity, non-intrusiveness, attention to users, locality of information, distribution of resources, emphasis on implementation and experimentation.

3.14.1 Simplicity

Despite the complexity of the design management problem, a conscious effort has been made to keep things as simple as possible. Conceptual support has been provided by Occam's razor ("*Entia non sunt multiplicanda praeter necessitatem*"), the well respected principle, in philosophy as well as in engineering, that gives preference to the simpler of two otherwise equivalent descriptions of a phenomenon.

The one sentence summary of this research could be that we propose the reduction of the

design management problem to the management of the design trace, the unifying element used to represent, document, and automate the design flow, and to assist the designers.

In comparison, other systems provide much of the same functionality using two or more separate subsystems. For example the MMS [3] uses LISP functions to automate the design flow, while the history of the design is documented in a separate ASCII file. A similar separation between history and flow automation can be found in Ulysses [11], and in the Task Manager [15].

The design trace is similar to a Petri net, but it is simpler because it has no cycles and no tokens. Other researchers [30, 6] have proposed the use of *extended* Petri nets, possibly attracted by their theoretically intriguing property of being equivalent to Turing machines [2], and therefore conceptually capable of modeling any algorithm. But such property of the extended nets is not exploited in [30] or in [6], just as many other properties of Petri nets have not found application in a DMS. VOV explores the hypothesis that a simple BAD graph is sufficient for design management.

The interface between VOV and the tools is also simple. The trace-based system does not need to know anything about the *tools*; it only needs to know something about the *tool invocations*: the command line, the working directory, the user name, the host name, and the list of inputs and outputs – information that is can be easily generated at runtime. By comparison, almost all other proposed systems require detailed descriptions of the tools and of their behavior.

The trace-based system does not require information about the significance of a place or about the meaning of running a tool. Places and transitions are treated as black boxes. All the knowledge specific to a design style is confined within the tools, thus eliminating a great deal of complexity from the DSM, and making it applicable in domains other than electronic design, that is wherever there is a set of automatic tools to be managed.

The quest for simplicity is also responsible for a number of concepts that are absent from VOV. Notable is the absence of any form of strong typing of either data or tools (see Section 2.3). Data are recognized only by database and by name within the database, not by value, meaning or type. There is no distinction between required and optional inputs of a tool, as in *decol* [34] or in the Task Manager [15], because inputs and outputs are declared at runtime and are all equally important.

Simplicity is key to the success of any system, because a simple system is easy to understand, use, and implement. A successful implementation, that is a detailed, robust, and friendly implementation, is the strongest promoter of any system among its users. It is essential for an ADM to have users, not only to validate the system, but also because the experience gained with use leads to a better understanding of the design management problem.

3.14.2 Non-intrusiveness

An early goal of this research was to develop a **non-intrusive** design manager that does not change the way designers use the tools. Tools should be easily incorporated in the system, and the designers should have **unrestricted access** to all the capabilities of a tool.

Since the trace is managed automatically by the tools, a designer can be unaware of the existence of VOV and yet benefit from its basic services, such as tracking of the design activity and conflict detection.

The integration of a tool into the system, whether through recompilation or through encapsulation, is simplified because the only objective is to provide communication with the server, while **maintaining** the look-and-feel of the tool. The objectives of encapsulation vary among the previously proposed DMS. For example, [24, 34, 15] require capsules to give every tool a **common** look-and-feel, by renaming tools and options according to a consistent scheme. EDMS [24] requires the encapsulation program to describe the type of the tool and of all its inputs and outputs, to perform pre and post-processing and to check the success of the tool. Such capsules are complicated and require effort to be written. For example the encapsulation of Verilog required 7 days. By comparison, the VOV capsule for Verilog has been written in about three hours.

3.14.3 Distributed resources, localization of information

The resources offered by modern design environments are distributed across many machines in the network; CPU cycles are available on several workstations and large mainframes, and data can be distributed among many file systems.

VOV exploits distributed resources by allowing slaves and designers to use any machine in the local network, as long as both the machine and the VOV server have visibility of the design data. The coordination of slaves is completely decentralized; slaves can be added and eliminated at any time, and the power and the resource list of each slave are determined by the slave itself. The design trace models concurrency and allows parallel retracing to be used whenever possible. The dispatching mechanism balances the load on the slaves to exploit the power of idle machines.

In contrast to the scattered distribution of resources is the need to maintain a localized distribution of information. All information related to a specific task should be organized so that it appears localized and easily accessible. A parallel can be found in the software domain: as a program that uses global variables is more difficult to maintain than one that uses only local variables, so a DMS based on a central database is going to be harder to manage than a DMS which

emphasizes distribution and localization.

In VOV, the information about inputs and outputs of each transition is generated locally by the tool itself, or by its capsule. Locality has also a temporal meaning in the sense that the information is generated while the tool is executing. If the behavior of a tool changes, the tool developer must update, at most, the capsule of that tool only, while the rest of the system is not affected.

Other DMS systems use centralized configuration files to describe the capabilities of the CAD system. Ulysses [10] uses the blackboard as the central global database, the NELSYS system [50] uses a centralized rule base to analyze the evolution of the design. In both cases, the addition or modification of a tool requires a complex change of the global information, to predict all the possible implications of that change on the system.

Cadweld [17] is close to achieve locality of the tool description, by means of its CAD Tool Objects (CTO), but some aspects in the CTO description require global information. For example, the “priority” or the “robustness” field, both used by CAD tasks to choose one of the CTO’s that volunteer for some operation, make sense only in a comparison between two CTO’s. Therefore, a developer of a new CTO needs to know the values of those fields for all potentially competing CTO’s in order to position accurately the new CTO in the global spectrum.

3.14.4 Focus on users

Understanding the users is a key element for the success of a software project. Paul Heckel, in his study on friendly software design [29], ranks “know your audience” near the top of the list of suggestions to the software developer, second only to the rather obvious “know your subject.” Similarly, Rubinstein and Hersh’s [44] first advice is: “Know thy user, for he is not thyself.”

The audience of a DMS consists of a range of potential users, each with different needs and expectations: expert and novices designers, project managers, system administrators and tool integrators, corporations.

A *novice designer* asks for a fully automated system that minimizes the learning effort. A novice designer wants to be led through the design and is probably willing to compromise on design quality. Their designs are likely to be routine rather than aggressive. An *expert designer* knows more about his design and his tools than he expects a computer to know; he does not want to be told what to do, but demands a docile and non-restrictive DMS. A *project manager* wants to

have control over the design activity, and wants to know what the designers have done. Sometimes, he also wants to enforce specific design methodologies that are deemed safe and robust. A DMS is valuable to a *corporation* when it reduces its vulnerability to the turn-over of designers, who may take away with them precious information and experience [45]. A DMS should therefore help improve the documentation of each design.

In VOV, the initial focus has been on the more challenging needs of the expert users. The attention has since been extended to novices with the development of the *assistant*. This approach has been effective; after investing more than a year (Jan 89 to Mar 90) developing the non-intrusive tracing and retracing mechanism, it was possible to develop the prototype of the assistant on top of the tracing mechanism in less than a month (June 1990).

Attention to the users requires emphasis on the user interface, the software layer that gives the users access to the services provided by the DMS. All the three types of interface mentioned by Rubinstein [44] have been implemented: command-based, menu-driven and graphical. The details are reported in Section 4.3.2. Of the DMS reported in the literature only EDA's system EDMS [24] offers all three types of interface.

3.14.5 Emphasis on team design

Team design is the norm in the electronic industry. Some authors [51] state that the correct approach to coordination of team design is to try to give each designer "the illusion that he/she is working on a single designer system." This is an arguable statement. Even a single designer can perform many tasks simultaneously, either on the same machine or on different machines in the network. The conflicts that can be generated by two interfering tasks are the same whether the two tasks have been initiated by the same designer or by two different designers. So the fundamental problem is not coordination of team design, but rather coordination of concurrent activities.

VOV emphasizes that it is possible and advantageous to work in parallel. Conflict detection protects the integrity of the data and avoids wasteful duplication of efforts. The design trace promotes cooperation among designers, because the activity of each designer is visible to all designers. The notion of ownership of parts of a design, a trademark of the workspace based systems (e.g. [15, 3]), is relaxed and it is conceivable that one designer might pick up an activity where another designer has left off.

VOV is a native multi-tasking system, not a single-user system extended to handle many users.

3.14.6 No restriction to data visibility

An important aspect of design concerns the **visibility** of the data. A human being prefers to work with just a few objects at a time; the visibility of other objects may be distracting and confusing. This is why people prefer to work with hierarchical rather than flat file systems, and to program with local rather than global variables.

A DMS should help focus the designer's attention on the objects of interest. In some systems [31, 3], the data is fragmented using the notion of workspace: one workspace per designer, and public workspaces for shared data. Each designer sees only the data in his workspace and none of the data in anybody else's workspace. This arrangement accomplishes the additional purpose of protecting the data in the public workspaces. However, we believe that this fragmentation is an obstacle to true cooperation among designers.

In VOV, protection is already provided by the conflict detection services, so that there is no reason to link visibility with protection. To encourage cooperation, data visibility is not restricted within a project, and all designers can, if they choose, inspect any piece of data represented in the trace. Data can be organized according to design tasks using UNIX directories and symbolic links.

A related issue is the visibility of the trace itself. The design trace is normally flat and large, and a designer rarely needs to look at the entire trace at once. The burden of filtering out uninteresting details should be placed upon the user interface. Instead of presenting the entire trace, the graphical interface should respond dynamically to the requests of each designer by showing only the details of a subset of the trace, while the rest of the trace should be shown as an abstraction.

This smart graphical user interface has not been implemented yet for two reasons: the objective complexity of its implementation, and the fact that the graphical interface is not critical for the operation of the system, which can perform all of its functions without the designer ever seeing the trace. A dumb graphical interface that presents the entire trace in full detail is nevertheless available. Feedback from users of the system has already indicated that designers are interested in looking at the trace, and that the dumb interface is not satisfactory.

3.14.7 Ignore design hierarchy

Hierarchy is normally introduced in a design for three reasons: to break down a large problem into smaller ones, to allow reuse of components, and to hide details. However, hierarchy does not unconditionally have all these advantages, and there are situations in which a flat structure is preferable to a hierarchical one. Management of the design flow may be one of these situations.

The human inability to cope with a large number of entities at the same time forces designers to apply Caesar's "divide et impera, divide-and-conquer" rule – to break the design of a large system into the design of smaller subsystems. This rule is often applied recursively with the result that many designs gain a multi-level hierarchical structure.

Hierarchy is not a static property of a design. As the design evolves, hierarchical levels can be flattened or added, to suit particular needs of designers or tools. For example, the hierarchical structure used to describe the behavior of a system may be inappropriate from the layout point of view.

The hiding of details is made possible not by the hierarchy itself, but by the capability of producing abstractions of the subproblems and of their solutions. Hiding details makes some tools more efficient, but only if the tools are capable of using information from the abstraction. For example, a placement tool operates at only one level of the hierarchy and it only needs information about the size and shape of the cells (the abstraction), rather than about the internal structure of each cell. For the same circuit other tools may need to retrieve the detailed structure of each component in the hierarchy, in which case hierarchy may be more of hindrance than of help. For example, the logic simulator *musa* must flatten the hierarchy of a circuit on the fly in order to function.

Reusability of components depends upon the regularity of a design. Regular designs, such as RAM's and ROM's, used hierarchy effectively, because few components can be reused many thousand times.

The importance of hierarchy and its relevance in the domain of data representation are no longer issues open to debate, because most modern CAD systems support hierarchical representations of design objects. But what is the role of hierarchy in design management?

In VOV, the design hierarchy is inconsequential. It is the tool's responsibility to understand the hierarchical structure of a design and to communicate to the system what use is made of the hierarchical information. A tool that uses abstractions, for example a placement tool, will declare as input only the abstractions at one level of the hierarchy, while a tool that traverses the hierarchy, for example a logic simulator, will declare as input all the components of the circuit, down to the lowest level. For VOV, the difference between the two tools is in the number of inputs and in their outputs.

Some previous work on design management believe that it is necessary to provide special support for hierarchical design. For example, Bretschneider et.al. [6] use specially annotated arcs in a Petri net to represent conditions that must be satisfied by all the components of a description of a circuit. Kozminski [34] embeds the circuit hierarchy in the UNIX hierarchy of directories and

relies on this structure to provide management services.

Only few researchers (e.g. Chiueh et. al. [15]) have considered hierarchy in the design flow, although it is clear that some form of hierarchy exists. For example, in the *Octtools*, the task of routing a macro cell chip is performed by running the tool *Mosaico*, which in turn calls a sequence of thirteen other tools (see Figure 3.24). From the users' point of view, it is advantageous to associate the task of routing a chip to a single tool, that is *Mosaico*. From a DMS point of view, there is a loss of detail (for example, loss of potential parallelism) in considering *mosaico* as an atomic transition. The approach in VOV is to operate with a flat description of the design trace, for maximum control, while the user-interface is capable of presenting a hierarchical view of the trace.

The advantage of using hierarchy in the design flow arises from the abstraction power of hierarchy, that is from the possibility of hiding details. Little advantage can be obtained by reusing a flow segment, as a subroutine can be reused in a program. Suppose that the "same" flow is used to process two independent design objects, A and B, and to be more concrete, assume that A and B are two chips to be routed by *Mosaico*. We could think of *Mosaico* as a subroutine with one parameter, and then apply the subroutine first to A and then to B, but this would not always be good design management. Some of the steps in *Mosaico* can take minutes or even hours, and one should avoid repeating those steps unless necessary. To do that, the DMS must maintain information about the progress of A and B through the steps in the subroutine; for each operation in the subroutine, the DMS needs to know when it was executed, if it was successful, and which data was used. But by keeping all this data around, the DMS effectively replicates the *Mosaico* subroutine once for A and once for B.

There is a tradeoff between efficiency and hierarchy: efficient design management demands a flat trace, while hierarchy is useful especially at the interface with the users. VOV offers support for hierarchy in the trace through the notion of collapsable sets, as described in Section 3.8.1.

Chapter 5

Experimental Results

The ultimate goal of an ADM is to increase the productivity of CAD systems, reducing the design time or the design cost, or improving the quality of the delivered product. The value of an ADM should be measured in absolute terms by the difference in productivity level with and without it, or in relative terms by comparing its productivity with some other ADM. Both tests are difficult to perform, because of a lack of formal definition of productivity, and because the comparisons are hindered by the impossibility to keep “all else” equal while switching the ADM.

However, it is possible to accumulate some statistics on the design process, such as the total time spent by the tools and the total number of tool invocations, separating successes from failures. These statistics are related to the design time, and they can provide insight about the designers’ style and the performance and reliability of the tool set.

In this chapter we present these statistics for a number of designs in which VOV has been used, including some projects developed by students of a VLSI design class.

5.1 Statistics on the design

VOV keeps a record of count and duration of all transitions, distinguishing the transitions invoked by the designers from those invoked by VOV itself as part of the retracing mechanism. Duration is measure in seconds of elapsed time, not in CPU time. Successful and failing transitions are counted separately, and for the failing transitions all 4 failure modes described in section 3.6 are considered: wrong exit status, invalid input, missing output, and interruption.

The following statistics are maintained automatically for each design trace:

TOTAL TRANSITION TIME (TTT) and COUNT (TTC):

TOTAL SUCCESSFUL TRANSITION TIME (TSTT) and COUNT (TSTC):

TOTAL FAILED TRANSITION TIME (TFTT) and COUNT (TFTC):

USER-INITIATED TRANSITION TIME (UTT) and COUNT (UTC):

AUTOMATICALLY-INITIATED TRANSITION TIME (ATT) and COUNT (ATC):

USER FAILED TRANSITION TIME (UFTT) and COUNT (UFTC):

USER SUCCESSFUL TRANSITION TIME (USTT) and COUNT (USTC):

AUTOMATICALLY FAILED TRANSITION TIME (AFTT) and COUNT (AFTC):

AUTOMATICALLY SUCCESSFUL TRANSITION TIME (ASTT) and COUNT (ASTC):

Some obvious relations between the quantities mentioned above are:

$$TTT = TSTT + TFTT$$

$$TTC = TSTC + TFTC$$

$$TTT = UTT + ATT$$

$$TTC = UTC + ATC$$

$$UTT = UFTT + USTT$$

$$UTC = UFTC + USTC$$

The conflicts detected during the design process are also counted:

CYCLES CONFLICT (CC): Number of transitions that attempt to define a cyclic dependency among design data.

INVALID-INPUT CONFLICT (IIC): Number of input conflicts.

OUTPUT-REDECLARING CONFLICT (ORC): Number of output conflicts.

LOCK CONFLICT (LC): Number of locking conflicts.

Another empirical measure of the quality of a design methodology is the ratio between the total duration of the valid transitions (VTT) and the total transition time TTT:

$$Q = \frac{VTT}{TTT}$$

A value of 1.0 for Q is an indicator of the “perfect” design methodology: each transition has been executed exactly once, and all transitions have been successful and useful. The more the transitions are retraced, the lower Q gets, because TTT increases and VTT does not. The value of Q fluctuates during the design process, as transitions are first invalidated (Q drops) and then retraced (Q rises). This measure makes most sense after the design has been completed. A slightly more stable measure is the *perceived quality* Q_p , which takes into account the transitions that are currently NOT VALID, and therefore do not contribute to VTT, but which are presumably going to be executed sooner or later. Let NTT be the expected cumulative duration of those transitions; then we define

$$Q_p = \frac{VTT + NTT}{TTT + NTT}$$

5.2 BRIC

BRIC is an integrated circuit for music generation conceived by Prof. John Wawrzynek of UCB. The design of this chip, which started in Spring 90 in an advanced VLSI design class, was the first real design managed by VOV. A collection of about 30 machines (SUN-4 and DECstation 3100) running the operating system SPRITE [42] was available for this design. A large set of tools were employed, including the *Octtools*, some commercial tools such as Verilog, and some new tools especially created for this chip. This was a challenge to VOV’s flexibility to encapsulate many diverse tools. The most troublesome tool to encapsulate was Verilog, mostly because its licence policy allowed it to run on only a few machines in the network, and because it required the use of a command line password that depended on both the machine name and the month. The encapsulation script not only described the input/output behavior of the tool, but also computed automatically the correct password.

The testing strategy involved a comparison of the logic simulation output with a behavioral model written in C. The trace therefore includes the compilation of the C file, the execution of the model with several input stimuli, the Verilog simulation with the same stimuli and the comparison of the outputs using the UNIX utility `diff`, encapsulated by the script `vov_diff`.

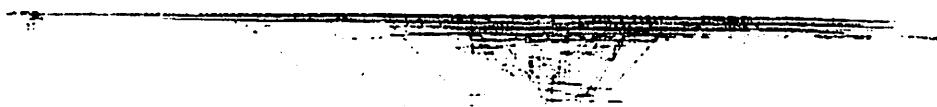


Figure 5.1: The trace for the BRIC project consists of more than 1200 places and 339 transitions. Large traces such as this tend to be relatively wide and shallow.

A team of eight students, including the writer, cooperated on the design. These students were experts, in the sense that they had already designed a chip using the *Octtools*. The reaction of the designers was our main concern: what did they think of VOV, did they like it? There was no negative response, but a general sense of acceptance. One clearly positive, even enthusiastic, response came from the student in charge of the pipelined multiplier array, who praised the automatic parallel retracing mechanism.

Although we were all working on the same trace, VOV registered no conflict due to concurrent activities of two designers. This was mostly due to a clear distribution of the design tasks and because of the spontaneous organization of the data into an ordered directory structure. In a couple of cases a designer continued a task from where another designer had stopped; the trace was the media to communicate unequivocally to the second designer what the first designer had done.

The BRIC trace is shown in Figure 5.1. The trace contains 1239 places and 339 transitions and 22 sets. 2141 tool invocations have been recorded, for a total of more than 105 hours of execution time. When this snapshot of the trace was taken, the perceived quality was $Q_p = 0.173$, meaning that on average all transitions have been repeated more than 5 times.

5.3 Floorplanning an FPU

The FPU project consists of the floorplanning of a floating point unit obtained from an industrial source. This is a single designer project, and its real objective is to test the floorplanning capabilities of the *Octtools*. The FPU trace is shown in Figure 5.2. The long tail in the trace corresponds to the tools in the *Mosaico* sequence, while most of the nodes in the top eight rows are part of the floorplanning activity.

The chip consists of 16 blocks, each with a fixed aspect ratio and a large number of floating pins, for a total of more than 3000 floating pins in the chip. The program puppy, used for this floorplanning, runs in-place because it changes the location of the pins in the master copy of

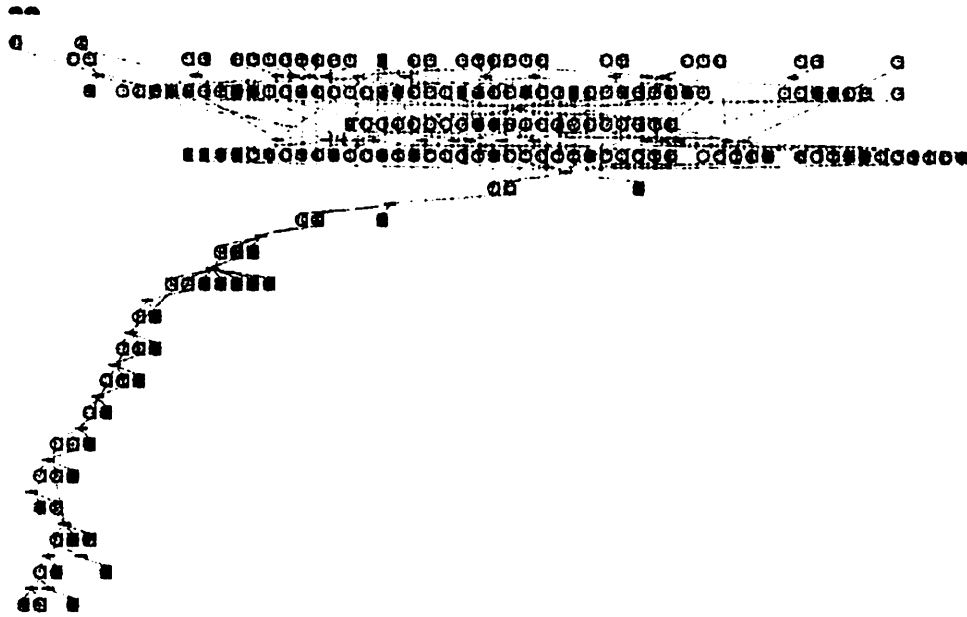


Figure 5.2: The trace for the floorplan of an industrial FPU.

each block in the floorplan; the masters are both inputs and outputs for *puppy*. After floorplanning, each block is processed by a module generator that tries to satisfy the constraints generated by the floorplanner. Upon successful generation of the modules, a new netlist is assembled and the chip is routed by *Mosaico*.

The trace consists of 16 branches, one for each block in the chip. These branches converge one first time into the *puppy* transition, then they proceed independently through the module generators and they finally converge again into the transition that assembles the floorplanned netlist. The double convergence of these 16 flows prevents any sensible partition of the trace into smaller units, so that the whole floorplanning task and its trace should to be dealt with as it is, flat and large. Other systems proposed in the literature do not seem to be able to deal with complex activities such as this.

5.4 Compilation of VOV

Another project that is used to test VOV is the compilation of VOV itself. VOV consists of six executables and 4 libraries, generated from about 27000 lines of C++ code, split into 131 files. The compilation trace includes 448 places and 122 transitions. To date, a total of 5289 transitions have been traced, for more than 171 hours of compilation and linking. The trace quality Q is 0.019,

a small number which means that the files have been compiled about 50 times since last March, when the accounting began. Parallel retracing is the main reason to use VOV for compilation.

5.5 VOV in a VLSI design course

CS250 is a graduate course on VLSI design regularly taught at UC Berkeley. The *Octtools* have been used in the class since the Spring of 1988 as a way to expose the students to current CAD technology and to enable them to complete a VLSI project of the complexity of a four-bit microprocessor. Traditionally the course is divided into two halves: the first half of the semester consists of a series of five laboratory exercises that show the capabilities of the *Octtools*, while in the second half the students, in groups of three or four, use the tools for the final class project.

VOV was introduced in CS250 in the Fall semester 1990, when the class was taught by Prof. John Wawrzynck and 25 graduate students were enrolled.

We felt that our moral responsibility was to give priority to the teaching of VLSI design rather than to using the students as guinea-pigs in our experiment on VOV. That is why we only recommended the use of VOV throughout the semester, but never enforced it. The results can be compared with previous offerings of the same class.

5.5.1 The laboratory exercises

The automatic assistant was the last piece of VOV to be developed, but it is the first one seen by a novice designer. Three of the five laboratory exercises (1, 2 and 5) were based on an example trace that was prepared by the teaching assistant (TA). Each trace was proposed to the students as an example of what they were supposed to learn in the exercise. The other two labs (3 and 4) had no example trace because they consisted mostly of hand layout using VEM.

The design environment consisted of a cluster of 24 color VAXstation 3100, each with 8Mbytes of memory, all connected to the same disk server. Students who had access to other more powerful workstations were allowed to use them.

The VOV tutorial, included in Appendix A, and a short twenty minute presentation on the philosophy of the design manager, was all that was given to the students to understand VOV. The tutorial leads the students through the design of a seven segment display driver, including automatic synthesis, layout, and simulation.

The first lab was simpler than the tutorial, because it covered only synthesis and simu-

```

vov_bdsyn 7SegDriver.bds
vov_misII -f script -T oct -o 7SegDriver:logic 7SegDriver.blif
vov_bdnet 7Seg.bdnet
vov_musa -i 7Seg.musa 7Seg:symbolic
# Total estimated duration: 1m56s.

```

Figure 5.3: The script corresponding to the example trace used in the first assignment. This trace refers to the synthesis of a seven segment display driver; the students had to synthesize a 4-bit ALU. Two variables substitutions were required: `7SegDriver` \rightarrow `bitslice` and `7Seg` \rightarrow `alu`.

lation of a logic network, without layout. The script of the trace prepared by the TA is shown in Figure 5.3. This example trace dealt with the synthesis of a seven segment display driver, while the students were requested to build a 4-bit ALU using a similar flow.

As in previous years, the first homework turned out to be easy. Many students used VOV successfully. Others ran into trivial technical difficulties, that were caused by problems with their UNIX account. A few of them quickly gave up the ADM preferring to use the tools without it. Of those who used VOV, a few did not understand the distinction between the example trace and their own design trace. This was because they did not have a chance to see their own trace, since the tutorial did not teach them how to access the graphical interface. Others successfully navigated through the menu-driven interface and discovered functions not described in the tutorial.

Many students did not understand the slave mechanism and did not know how to monitor the activity of the slaves. A common operation that students wished to perform was the elimination of transitions from the trace, but they were unable to do it because they had not been taught how.

The students had to overcome a difficulty with the VOV assistant, which in that period allowed only one variable substitution, while the example trace would have required two (Figure 5.3). Some students executed the assistant twice; others built the trace by executing the transitions manually.

The trace was particularly useful to enable the TA to help the students in difficulty, because it recorded precisely what the students had done and made the diagnosis of the problem much easier than in the past.

For completeness we report the technical problems encountered with VOV in the first homework. Due to a project name conflict one student could not start her server on the machine she had chosen, and had to ask for assistance to understand what was happening. Some servers entered an infinite loop caused by a bug that was discovered and fixed only later in the semester.

Those servers had to be killed and restarted by hand. A few times, due to incomplete setup of the students' UNIX accounts, no slaves were connected to the server and any retracing command would hang, waiting forever for a slave to become available. Some students were not familiar with the notion of a server/client architecture and did not know how and when to kill the various VOV processes. The capsules `vov_bdnet` and `vov_musa` were found to be incorrect.

The data collected during the first laboratory are shown in Table 5.1. The large variance is an indication that the students have different ways to approach the tools. For example, student `aa`'s data shows only four transitions, suggesting that he must have turned VOV on and off, because much of his work did not get traced. At the other extreme, student `ak` used VOV throughout the process for a total of 89 tool invocations, but he did not take advantage of the automatic retracing mechanism, because only 9% of his tool executions were done automatically. Student `wb` had 76% of his tools execute as part of automatic retracing. These results prompted the need for a quick lecture on the retracing mechanism, which was better exploited in the second homework, as shown in Table 5.2.

The second laboratory dealt with various optimization techniques using `misII` and with the layout tools `wolfe` for standard-cells and `octpla` for PLA's. The example trace for this lab is shown in Figure 5.4. As far as VOV is concerned, this lab was in large part a replay of the first, except for some improvements in the understanding of the system and some more users giving up VOV, apparently because of problems caused by the limited disk space made available to the students.

The third and fourth homeworks dealt with symbolic layout of an elaborate 4-bit datapath. These labs were not based on example traces, and there was no particular need to use VOV. Some students with access to powerful machines elected to use it anyway, asserting that it helped them maintain the consistency of their hierarchical layout.

In the fifth exercise the students assembled a complete chip using the datapath prepared in lab 4, the seven segment display driver used in lab 1 and an automatically synthesized controller. The chip, complete with bonding pads, multiplies two 4-bit numbers and displays the product on a pair of seven segment displays. The exercise required the use of several CPU intensive tools to place and route the chip, including `puppy` and `Mosaico`. The complete trace of the design of a similar chip was prepared by the TA was offered to the students as example.

The environment in which most students were forced to work was too restrictive; a disk quota of 2.25Mbyte per student forced the students to a difficult bookkeeping to maintain enough disk space so that the tools could run. The final chip alone required more than 1 Mbyte. VOV was

Results from Lab 1										
Student id	aa	ll	al	ak	aq	as	bc	cm	wb	
Total Trans Time	98	717	1030	2793	347	594	664	305	1346	
Total Trans Count	4	45	38	89	23	27	32	20	32	
User Tt	79	499	963	2546	227	580	269	195	319	
User Tc	3	16	35	75	13	25	8	14	10	
User Success Tt	76	140	827	1100	167	277	266	193	197	
User Success Tc	2	9	28	33	8	12	7	13	5	
User Failed Tt	3	359	136	1446	60	303	3	2	122	
User Failed Tc	1	7	7	42	5	13	1	1	5	
User Failed Wrong Status Tt	3	41	136	815	36	103	3		61	
User Failed Wrong Status Tc	1	3	7	38	4	6	1		2	
User Failed Generic Tt		314		593				2		
User Failed Generic Tc		3		3				1		
User Failed Defective Output Tt						200			8	
User Failed Defective Output Tc						7			1	
User Failed Invalid Input Tt		4		38	24				53	
User Failed Invalid Input Tc		1		1	1				2	
Auto Tt	19	218	67	247	120	14	395	110	1027	
Auto Tc	1	29	3	14	10	2	24	6	22	
Auto Success Tt	19	199	67	199	112	8	371	110	772	
Auto Success Tc	1	25	3	10	8	1	20	6	15	
Auto Failed Tt		19		48	8	6	24		255	
Auto Failed Tc		4		4	2	1	4		7	
Auto Failed Wrong Status Tt		17		43	5	6	20		255	
Auto Failed Wrong Status Tc		3		5	1	1	2		7	
Auto Failed Generic Tt										
Auto Failed Generic Tc										
Auto Failed Defective Output Tt		2					2			
Auto Failed Defective Output Tc		1					1			
Auto Failed Invalid Input Tt				5	3		2			
Auto Failed Invalid Input Tc				1	1		1			
Conflict Redeclaring Output	2	9	33	72	11	25	5		3	
Conflict Invalid Input										
Conflict Cycle										
Conflict Lock										

Table 5.1: The statistics collected by VOV for the first laboratory exercise show a large variance in the way students have used the tools.

Results from Lab 2						
Student id	hc	aa	aj	au	cm	ll
Total Trans Time	1625	7266	6013	501	16998	11633
Total Trans Count	16	35	49	11	164	182
User Tt	441	4969	4584	421	4720	3409
User Tc	9	15	28	6	74	46
User Success Tt	379	4252	2812	409	3103	2814
User Success Tc	7	13	20	5	47	42
User Failed Tt	62	717	1772	12	1617	595
User Failed Tc	2	2	8	1	27	4
User Failed Wrong Status Tt	62	717	35	12	213	
User Failed Wrong Status Tc	2	2	2	1	2	
User Failed Generic Tt						545
User Failed Generic Tc						2
User Failed Defective Output Tt			15		247	50
User Failed Defective Output Tc			1		11	2
User Failed Invalid Input Tt			1722		1157	
User Failed Invalid Input Tc			5		14	
Auto Tt	1184	2297	1429	80	12278	8224
Auto Tc	7	20	21	5	90	136
Auto Success Tt	1184	2297	1158	41	11233	7456
Auto Success Tc	7	20	15	2	71	130
Auto Failed Tt			271	39	1045	768
Auto Failed Tc			6	3	19	6
Auto Failed Wrong Status Tt			267		741	40
Auto Failed Wrong Status Tc			5		12	1
Auto Failed Generic Tt					6	
Auto Failed Generic Tc					1	
Auto Failed Defective Output Tt			4	21	64	
Auto Failed Defective Output Tc			1	2	2	
Auto Failed Invalid Input Tt				18	234	728
Auto Failed Invalid Input Tc				1	4	5
Conflict Redeclaring Output		14	21	3	46	15
Conflict Invalid Input						
Conflict Cycle						
Conflict Lock					1	3

Table 5.2: The results of the second laboratory exercise confirm the large variance in the way students use the tools, as observed in the first exercise. A better use of the retracing mechanism is apparent.

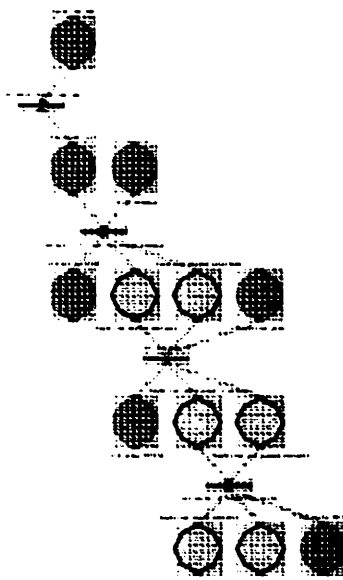


Figure 5.4: The example trace for lab 2.

one of the early victims of this cleanup effort, because even the 300kbyte or so used for the trace and the journal were precious. VOV could, in principle, use the trace to determine which data are essential and which could be deleted to reclaim disk space, but in practice such service was not available at the time because its importance had been underestimated. The students with access to private workstations did not have space limitations, and run VOV without problems.

Lab 5 turned out to be relatively easy, mostly because the tools involved were quite robust. By the end of the seventh week, most students produced a complete working chip, which could have been sent out for fabrication. We also observed that the students were using the tools in a more sophisticated mode than ever before; they were exploring esoteric options and stretching their capabilities.

In comparison, the fifth assignment in the previous offering of the class (Fall 89) involved the assembly of a multiplier by connecting a 4-bit alu and a PLA, with the storage elements provided off-chip, resulting in a much simpler and smaller chip.

Each lab was an opportunity to debug VOV and to tune its performance. For example, lab 5 exposed an infinite loop bug related to the management of tools that run in place and excited only by a particular sequence of tool failures. The excessive overhead that frustrated many students who used VOV on the smaller machines was traced back to a timing interaction between some clients and the paging mechanism in UNIX, which delayed the job-dispatching routine in the server by up

to several minutes.

5.5.2 The final project

For the final project the students could choose between hardware, software or architecture. Of the 25 students, 4 chose a software project, 8 chose the architecture project, and the other 13 split into 4 groups, each designing a VLSI chip. Thus, there were only four VLSI projects that could take advantage of VOV; two did.

The two groups that did not use VOV used the LAGER CAD system [14], because they were already familiar with it. One group designed a “low power color space translator,” a digital chip with about 6000 transistors, the other group produced a 5000 transistor chip for bus arbitration in a particular multiprocessor architecture. The groups’ own estimates of design time range between 260 and 280 hours per person.

Another three person group designed a “medium access control chip” (MAC chip) for interprocess communication. They produced two complete versions of the chip, one using `wolfe`, the other containing some optimized hand layout. Each version had about 31,000 transistors, of which about 18,000 belonged in a RAM. They said that VOV was helpful in managing and processing the more than 30 BDS behavioral descriptions of the circuit components, but they also preferred to run some tools directly to avoid the capsule overhead. They used two servers because they had access to two clusters of machines. They estimated their design effort to be at least 400 hours per person. The statistics from their two traces are shown under MAC in Table 5.3.

The last design group was formed by four students and developed a 9-band “spectrum analyzer for audio signal” with video output. The challenge for this chip was the use of experimental software for layout of analog circuits such as switched capacitor filters. Most of those tools were not encapsulated. In order to fit within the tiny-chip MOSIS frame, the group separated the analog part from the digital part and produced two separate chips that communicate on a 6 bit bus. The 4150 transistor count for both chips is relatively small, because of the few transistors in the analog part.

This group also used VOV, not for the entire design, but rather for the final assembly of the two chips, the most automated part of the design. An interesting episode happened with the compactor `sparcs`, which on one chip consistently crashed on the third iteration. In order to overcome the problem, the students discovered that they could reliably obtain three iterations by running `sparcs` twice, the first time for two iterations, the second time for just one. This was an

example of inventive design methodology that could not have been planned a-priori, simply because nobody knew that `sparks` could behave so strangely. The trace based system allowed the students to achieve their goal as soon as they discovered how to do it.

The students estimated their design effort to have required about 250 hours per person. The statistics captured by VOV are shown under SAAS in Table 5.3.

5.5.3 Comment

All four chips designed in this class were more complex than in the past, and the two chips managed at least in part by VOV were more complex than the other two.

Although this increase in student productivity observed in the labs and in the final projects was precisely the effect we hoped for when we introduced VOV in the class, we are not in the position to single out VOV as the one element responsible for it. We must instead mention other factors that could have played major roles, namely the improved computing environment, the fact that the labs were designed to avoid all tools known to be unreliable, and the extraordinarily competent support offered by the TA.

Students who did not use VOV suffered the usual problems seen in previous semesters: they forgot to run tools (mostly `vulcan`), they unnecessarily reran tools, and their concurrent activities conflicted. The main justification to not using VOV was the overhead in running the tools.

The students who used VOV expressed satisfaction for the system. Occasionally they preferred to explore some design methodology without VOV, that is by calling the tools directly, in order to avoid the capsule overhead. Then, when they were sure to understand the tool flow, they invoked the encapsulated tools to store the flow in the trace. This is not the behavior that was expected while developing VOV, but it is certainly allowed, and it shows the importance of having capsules that maintain the behavior of the tools.

The main lesson from this experience is that it is necessary to reduce the overhead due to the encapsulation process. There does not seem to be much space to improve the speed of the interpreted shell scripts currently used to encapsulate the tools. On the other hand, the compilation option, described in Section 3.4.1, has yet to be fully explored, and it promises the reduced overhead.

Results from the final projects			
Project name	MAC1	MAC2	SAAS
Total Trans Time	34h	10h48m	4h02m
Total Trans Count	592	129	152
User Tt	19h	10h25m	1h58m
User Tc	194	85	56
User Success Tt	5h36m	4h10m	51m23s
User Success Tc	182	79	43
User Failed Tt	13h	6h15m	1h07m
User Failed Tc	12	6	13
User Failed Wrong Status Tt	2h30m	3m49s	6m15s
User Failed Wrong Status Tc	2	1	6
User Failed Generic Tt	21m30s	6h10m	0
User Failed Generic Tc	4	2	0
User Failed Defective Output Tt	37s	51s	5m03s
User Failed Defective Output Tc	1	3	4
User Failed Invalid Input Tt	10h20m	0	57m47s
User Failed Invalid Input Tc	5	0	3
Auto Tt	15h15m	23m28s	2h04m
Auto Tc	398	44	96
Auto Success Tt	11h30m	21m33s	1h26m
Auto Success Tc	372	40	63
Auto Failed Tt	3h45m	1m55s	38m08s
Auto Failed Tc	26	4	33
Auto Failed Wrong Status Tt	2h30m	1m55s	10m11s
Auto Failed Wrong Status Tc	12	4	12
Auto Failed Generic Tt	2m02s	0	2m38s
Auto Failed Generic Tc	2	0	3
Auto Failed Defective Output Tt	1h03m	0	4m54s
Auto Failed Defective Output Tc	5	0	8
Auto Failed Invalid Input Tt	8m45s	0	20m25s
Auto Failed Invalid Input Tc	7	0	10
Conflict Redeclaring Output	28	3	13
Conflict Invalid Input	2	0	0
Conflict Cycle	1	0	0
Conflict Lock	0	0	0

Table 5.3: Statistics for the final projects that used VOV.

Chapter 6

Conclusion

The use of design traces for management of the design activity is the main proposal presented in this dissertation. The trace, a bipartite directed and acyclic graph, is built and managed dynamically using information provided by the tools at runtime; the trace captures both the history of the design and the data dependencies. In the process of managing the trace the system protects the integrity of the design data, by detecting situations in which a tool is about to destroy or overwrite important data. The mechanism of retracing, the automatic repetition of the transactions represented in the trace, can be used to maintain the consistency of the design data. The server/client architecture of the system allows for a coordinated cooperation of concurrent activities, by one or by many designers.

The trace is first of all a response to the needs of expert designers. The trace answers the difficult questions about design. It is concerned about the details of the design activity. Other systems are concerned about the “high level” and stumble over the detail. Our trace based system allows an unrestricted access to the tools. Its fundamental feature is that it is non-intrusive. This is best demonstrated by the fact that the designers’ activity is unaffected by the system being turned on or off. The designers see the same tools and work in the same familiar environment (e.g. the UNIX shell). Of course, if the management system is turned off, no services are available, neither tracing, nor automatic retracing, nor coordination of concurrent activities. But even without these automatic services, design can continue.

For the large audience of novice designers, the traces constitute a precious set of well documented examples of how tools can be sequenced to achieve a particular goal. An automatic assistant helps the designers extract information from an example trace and apply it to the current design problem. A library of traces can be assembled to describe many of the routine activities pos-

sible within a CAD system. Example traces show the tools used in context, giving a more concrete meaning to the tool capabilities, which were previously available through the sterile medium of abstract manual pages. Through examples a novice designer can easily learn to use very complex features of some tools. For example, while a command line such as

```
padplace -a -u _cFamily -D chip.pads -o chip:with_pads chip:symbolic
```

seems overwhelmingly complex if presented by itself, it is much more clear if presented in the context in which it is likely to be used. The command line means that the tool `padplace` should take the chip in the OCT facet `chip:symbolic`, and produce an output facet called `chip:with_pads` (option `-o`), in which all the formal terminals have been given an implementation using the pads in a particular pad family (option `-u _cFamily`). The pads should be positioned as described in the file `chip.pads` and laid down within the perimeter specified in `chip:symbolic:interface` (option `-a`).

Other researchers [15, 34, 3, 17] direct their energies towards a particular form of *tool encapsulation* which separates the tool invocation from the intention of the designer. This resolves in either a redundant renaming of tools and their options or in an elimination of some options which do not fit in the mental scheme of whoever is doing the encapsulation, thus effectively amputating the capabilities of the tool.

As opposed to virtually all other proposed design management system, our trace based system requires no a-priori description of the tools and of their capabilities. The only requirement, that a tool be able to inform the server about its inputs and outputs, is easily achieved either by linking the tool against a special library or by providing a capsule, normally a shell script that emulates the tool's input/output behavior. In either cases the modifications are localized within the tool, and can be done independently of all the other tools in the CAD system. In this way, new tools can be added to the design flow by the designers themselves.

A prototype of our DMS has been implemented and tested by a number of designers, both expert and novices. The feedback from users has been essential to identify many relevant issues in design management, such as the key role of the user interface, the need to be concerned about keeping the DMS small and responsive. Many traces have been generated, and many statistics have been collected on the number and duration of tool invocations. The statistic discriminate between automatic and manual invocations, and between successful and failing termination. The large variance in the statistics is the first hard evidence that each designer has a very personal style, some favoring a lot of manual interaction with the tools, others relying more on the automatic

retracing mechanism.

Our experiment in design management continues, strengthened by the direct experience and by the users' feedback. Research is under way to complement the system with a new tool to do statistical analysis of the performance of the tools in a CAD system, finding correlations between some measurements performed on the input data of a trace and some other measures on the corresponding outputs of the trace. It is hoped that these correlations can be used to provide statistical estimators of the performance of the toolset.

Automation of design management is a part of the CAD framework problem, so that the following aphorism applies [28]:

(...) there will never be a "right answer" to the CAD framework problem, only good answers and better answers.

VOV has been proposed as a good answer; I hope it will help find, sometime soon, a better one.

Appendix A

Tutorial

The objective of this tutorial is to help you become familiar with VOV, a design management system developed at UC Berkeley and integrated with the Octtools. In a simple exercise, you will use some of the Octtools to layout a combinational logic circuit starting from a behavioral description of the circuit. No knowledge of the Octtools is required. You will describe the behavior of a seven segment display driver (SSDD) using the language BDS, convert the description into logic equations using `vov.bdsyn`, optimize the logic equations with `vov.misII`, simulate the description with `vov.musa`, implement the layout with `vov.padplace` and `vov.wolfe`; all this under VOV's supervision, and with VOV's assistance.

The expected duration of the exercise is about an hour, but you can suspend it at any time and continue later.

A.1 Introduction

VOV is a design management system that provides many services: coordination of team design, history tracking, design data monitoring, data dependency analysis and others.

VOV non-intrusively monitors the activity of individuals as well as teams of designers, and maintains a record of each transaction invoked by the designers. All the design activity is captured in the **design trace**. The design trace is a bipartite directed graph, in which nodes are called either "places" or "transitions." Places represent design data, while transitions represent tool invocations, also referred to as "CAD transactions." Each transaction is characterized by a set of input places and a set of output places.

The trace can also be interpreted as a data-dependency graph. All the outputs of a transition are dependent upon the inputs of that transition. If any of the inputs changes, the transition must be repeated, that is, it must be "retraced."

A useful feature of VOV is its ability to automatically retrace (re-execute) a design sequence by using a previously generated design trace. This provides something like an automatic make facility, with the difference that you do not have to write any `makefile`. Instead, you simply have to execute a transaction once; after that, VOV knows when the transaction should be repeated and can repeat it automatically.

A.2 TUTORIAL: Design of a seven segment display driver

A seven segment display driver is a circuit with one 4-bit input called `data<3:0>` and 8 outputs `a, b, c, d, e, f, g, dp`. The top segment is called “a”, then, if you move clockwise, you find segments “b” “c” “d” (the bottom) “e” and “f”. The middle horizontal segment is “g”. The decimal point is usually called “dp”. The input is interpreted as an hexadecimal digit, while each of the outputs controls the corresponding segment of a seven segment display unit, so that the segment is lit if it is needed to represent the input number. The outputs are active high.

A.2.1 Start mini-VOV

Because you will only need a few of the functions of VOV to complete this exercise, you can use the “mini” version of VOV.

Make sure you have both `~octtools/bin` and `~octtools/bin/vov` in your path. If they are not, please edit your `~/.cshrc` file and modify your path variable.

Decide a name to describe the activity you will perform to complete this exercise. The name is important, because it identifies your work from that of other people who might be working on the same machine. The project name can be any alphanumeric string with no spaces. Examples are: `cpu badge microprocessor goophy xyz99`. It is a good idea to use a name which includes your initials, or your login name. Suppose that you choose the name `ACTut`.

Move in whatever directory you plan to do your work. For example it could be `~/vovtutorial`. There you should type:

```
mkdir ~/vovtutorial
cd ~/vovtutorial
vov_mini ACTut
```

The `vov_mini` command starts what is called the “vov server,” a program that runs in the background and monitors the activity of the tools you invoke. To get information from the server use the program `vov_sh`. For example,

```
vov_sh -I
```

asks the server to provide information about the status of the design. Like almost all the tools in the Octtools, `vov_sh` with no options has the effect of producing a usage message, with all the options understood by the tool. Try this, even if at this point the number of options of `vov_sh` might appear overwhelming.

Using `vov_sh -I` will also show you if the server is up and running. If nothing happens within a minute, or in the unlikely event that `vov_mini` fails, you should choose another name for your activity and try again.

Every time a tool is executed, whether it is invoked by you or by VOV, the tool connects to your server and declares all of its inputs and outputs. The server checks that all inputs are up to date and that the outputs can be overwritten. If that is the case, the tool proceeds with its task, otherwise a conflict has been detected and the user is queried to decide on how to resolve the conflict.

For this exercise, it is best if you work only from the shell that `vov_mini` gives you. If you want to open another window (even if running on another machine) and use that window as

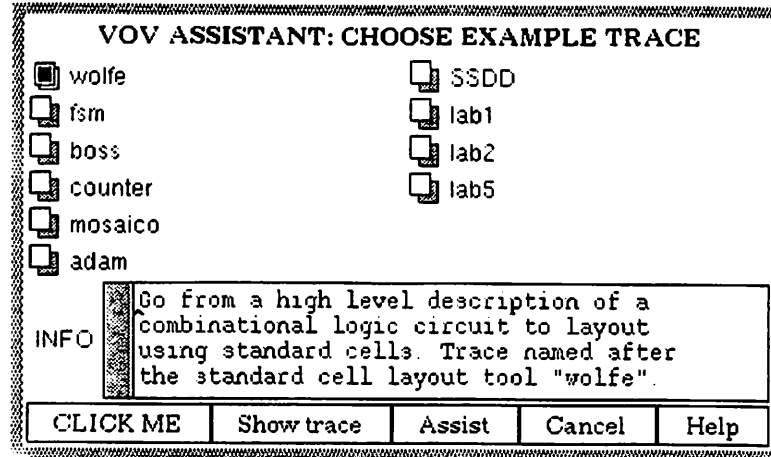


Figure A.1: The first pop-up dialog from the assistant. The trace called fsm has been selected, and the INFO field shows a short description of the trace. The designer can take a look at the trace (Show trace) or decide to use this trace as example (Assist).

well, you can do it, provided that you set the two environment variables `VOV_HOST_NAME` and `VOV_PROJECT_NAME` to the appropriate values. This can be done easily with an alias created by `vov_mini` called the same as the project name.

If the server is running, let's begin our design. If you knew the Octtools, you could start using them as if VOV did not exist. Thus you would probably write a BDS file with the description of the SSDD and then you would invoke the various tools `vov_bdsyn`, `vov_musa`, `vov_misII`, `vov_wolfe` in the right order, with the proper command lines.

But, let's assume that you do not know much about the Octtools. You need help, and you wonder if anybody else has ever done anything similar to what you have to do.

A.2.2 Enter the assistant

The steps that transform a behavioral description of a circuit into layout have been performed many hundreds of times by many expert designers. Each time, the tools have left a trace of their execution. Some traces have been saved and installed in a library. The program `vov_assist` can be used to extract information from such existing traces.

For this exercise, it is best to use `vov_assist` in its interactive mode:

```
vov_assist -i
```

You will get a pop-up dialog, with a list of all the example traces currently present in the VOV library (Figure A.1). The assistant is designed to be self explanatory. Make sure to click the buttons labeled `CLICK ME` and "Help" (or Ctrl-h) whenever you do not know what to do. For this exercise, here is the hint: choose the trace labeled `wolfe`.

Use of dialogs

The operation of the dialogs in VOV is mostly done with the left button of the mouse to select items in lists and to click on control buttons. The scroll-bar widget uses all three buttons, in a rather intuitive way: the left button moves the bar to the left, the right to the right, the middle allows you to drag the bar continuously.

The left button is also used to select text: you just have to click on it from two to five times. Two clicks select a word, three select a line, four a paragraph, five the entire text. One click simply moves the cursor. The selected text can be deposited into another window using the middle button.

If there are several edit fields in a dialog, you can use Tab and Meta-Tab to move from one to the other. Some control buttons have “accelerators,” that is you can type a key rather than pointing and clicking. In VOV, the *Help* button is accelerated by both Ctrl-h and by the Help key. *Cancel* and *Dismiss* are accelerated by Meta-Del.

A.2.3 The graphical interface

It is interesting to see what the trace looks like. You can do this by clicking `Show trace`. This starts the editor VEM on the trace, and an associated process, called an “RPC application,” that lets you browse the trace. At this point you are probably not familiar with either VEM or RPC. This is not a big problem if you are a bit adventurous and use this hint: Clicking the middle button on a VEM window pops up the VEM menu, while clicking the middle button with the “Shift” key pops up the RPC menu. The most useful commands have a “key binding” for quick invocation. The command you need is `view`, which is bound to the lower case `v`. To quit VEM use Ctrl-d in the console window.

The trace (see Figure A.2) is a bipartite directed graph with “places” which represent data, and “transitions” which represent tools. Each transition has some inputs and outputs. Move the mouse over the trace and view some of the nodes. Notice that at the top there is a place which is a behavioral description of a counter, while at the bottom there is a place which represent the layout of the counter. It is the purpose of this exercise that you learn what each of the intermediate steps does.

A.2.4 Getting assistance from the assistant

Now go back to the `vov_assist` dialog and click `Assist`. You will get another dialog, with more help (Figure A.3). Please spend some time to understand this dialog.

This dialog contains a script, consisting of all the command lines that have been used to generate this example trace. This is an alternative way to represent the trace. The trace has been fully annotated to help you understand it. To see some of the notes, click on the `Toggle notes` button. Please notice that the first transition to be executed was `vov_bdsyn counter.bds`. Why did the designer run this tool? What other tools were used? Why was `vov_wolfe` used?

You will soon notice that the example trace refers to a counter. In fact, all the files mentioned in the trace are called `counter.someSuffix` and all the OCT facets are called `counter:someViewName`. The assistant uses some heuristics to determine the “topic” of an

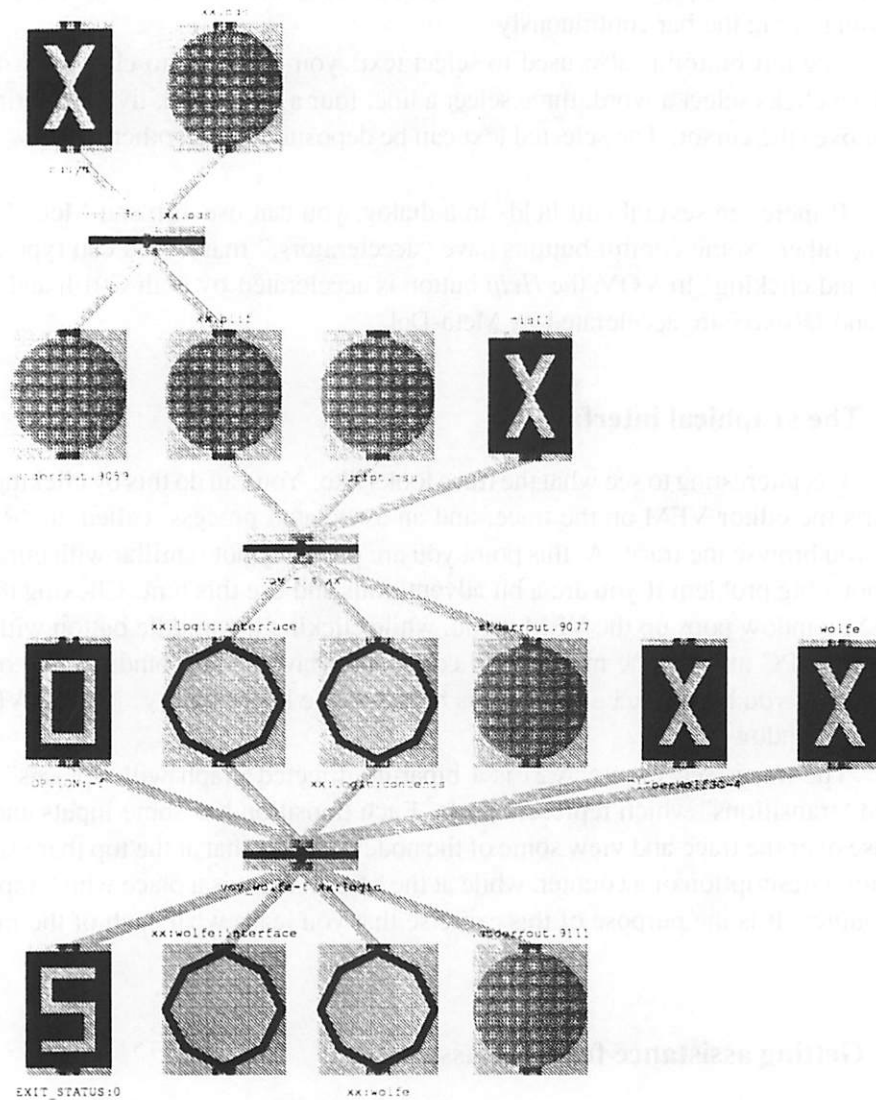


Figure A.2: This example of a trace should be read from top to bottom, inputs are on top, outputs on the bottom. Three transactions are represented in this trace. Although all the places are treated similarly, they are represented graphically by different icons depending on the type of the place: circles represent UNIX data files, octagons are used for oct facets, X's denote executables, O's command line options and S's exit status of transactions.

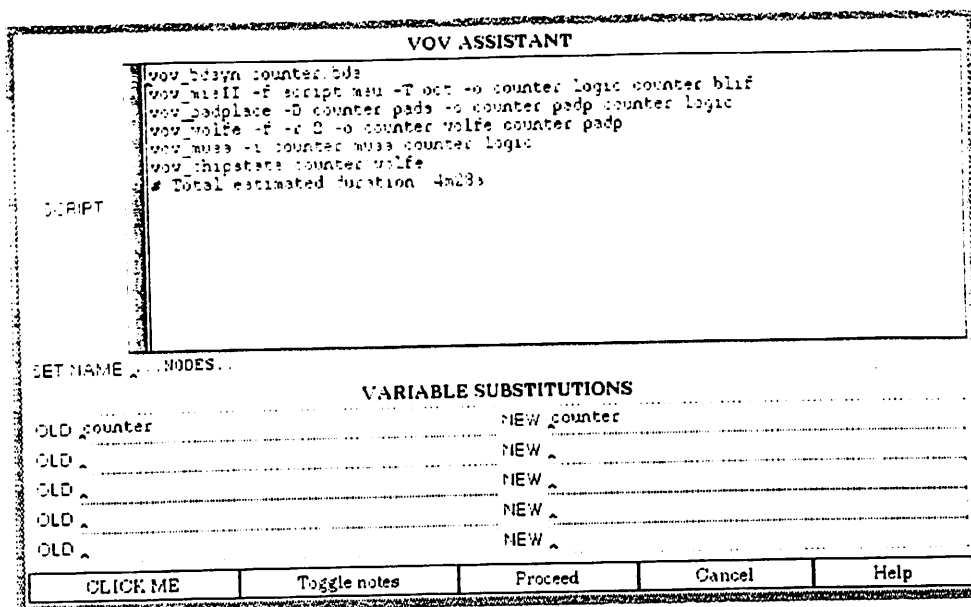


Figure A.3: The second dialog from the assistant.

example trace. For this trace, the topic is described by the string `counter`, which appears in the dialog next to the `OLDROOT` label.

You want to design a SSDD, not another counter. Thus, it is better to rename the files involved in the trace. Instead of `counter.something`, you might want to call your files `7seg.something`. Write `7seg`, or the new name of your choice, in the `NEWROOT` box and click on `Proceed`.

You will get yet another menu, titled `VOV ASSISTANT: COPYING A TRANSITION`. This is the first transition that you have to perform to go from a BDS description to layout: `vov_bdsyn 7seg.bds`. The annotations below the transition explain why you want to run this tool. Please read the annotations. Since you probably trust your assistant, you should simply read the information in the dialog, and click `Copy`. This means that the transition in the example trace is copied into your own trace, with the appropriate modifications to the command line. `vov_bdsyn` is not actually run at this point, but now VOV knows that you want to execute this transition, sooner or later.

Now another dialog has popped up. This says `COPYING A PRIMARY INPUT`. In order to run `vov_bdsyn` you need a file called `7seg.bds` which does not yet exist. You could very well write it from scratch, but here the assistant gives you an opportunity to get a good start. It offers you the possibility to copy the description of the counter into the file `7seg.bds`. At least, you get a syntactically correct file that should be relatively easy to modify later with a text editor like `vi` or `emacs`. Again, click on `Copy`.

Keep going: you will get about ten dialogs, in each of which you will have to click the `Copy` button. Eventually, you get a dialog that informs you that the transfer of information has been completed. Dismiss the last dialogs in `vov_assist` and go back to your shell.

For future reference

You could have obtained the same result you have just obtained by simply saying:

```
vov_assist -p wolfe -N 7seg
```

This batch use of the assistant will be useful if you want to use the example trace to process another cell. You will still use the dialogs if you want to copy only a part of an example trace.

A.2.5 Your turn to act intelligent

The assistant has done the best it could to help you, but it is not smart enough to do the design for you. You still have to make an effort to read some manuals and to understand the steps involved in your design. You won't have to read any manual to complete this tutorial, but you are encouraged to take a look at them soon, so that you can become a "power-user" of the Octtools.

At this point, the assistant has transferred into your design trace a modified version of a design flow which was used by some expert to build a counter. The SSDD can be built with a very similar flow. Also, the assistant has given you a good start on all the primary inputs that you need to design the SSDD. But now it is up to you to modify those files so that they contain meaningful information. Suppose that you have chosen the root name "7seg" for your files. In your working directory you will find the following files: `7seg.bds` `7seg.musa` `7seg.pads`. By now, you should know the purpose of each file, and you should modify them appropriately.

You should describe the behavior of the SSDD in the `7seg.bds` file. Then, you must also specify the position of the terminals in the layout, say inputs at the bottom, outputs on top, in the file `7seg.pads`. Finally, you must provide a complete simulation script to be used with `vov_musa`, by modifying `7seg.musa`. Figures A.4, A.5 and A.6 offer a very clear suggestion on how to do this. It would be nice if you could do this exercise without copying the figures¹.

A.2.6 The trace as a dependency graph

From the trace, we know that the final layout is going to be an OCT facet called `7seg:wolfe`. Even if it does not even exist yet, the server already knows how to obtain it. Try

```
vov_sh -h 7seg:wolfe
```

to get a short history of this piece of data. The use of the term "history" might sound strange at this point: how can the design have a history if we have done nothing, if no tool has ever been run? In fact, the term history is not being used properly. Instead we should say "dependency." `7seg:wolfe` depends upon all the transitions shown by `vov_sh -h 7seg:wolfe` (Figure A.7).

Try also

```
vov_sh -h 7seg.blif
vov_sh -h 7seg.bds
```

to see the history of other places.

The trace can be interpreted as a dependency graph. By default, with the `-h` option of `vov_sh`, you get a report on the visited transitions, four levels backwards, that is looking at the nodes "above" the one you query about. Using the `-H` option, you can also ask a report on all the

¹If you are lazy, the files are in `octtools/lib/vov/DATA/SSDD`.

```

!! Description of seven segment display driver.
MODEL "7seg"                !! Declare the name of this description.
    a,b,c,d,e,f,g,dp        !! Comma separated list of outputs.
    =                        !! The '=' is the separator between outputs and inputs.
    data<3:0>
    ;                        !! Terminator of I/O list.
CONSTANT
    Xzero  = 1111110#2,      Xone   = 0110000#2,
    Xtwo   = 1101101#2,      Xthree = 1111001#2,
    Xfour  = 0110011#2,      Xfive  = 1011011#2,
    Xsix   = 0011111#2,      Xseven = 1110000#2,
    Xeight = 1111111#2,      Xnine  = 1111011#2,
    XA     = 1110111#2,      XB     = 0011111#2,
    XC     = 1001110#2,      XD     = 0111101#2,
    XE     = 1001111#2,      XF     = 1000111#2;

STATE out<6:0>;              !! Intermediate signal.
ROUTINE main_routine;        !! Only one routine in this description.
    dp = 0;                  !! Decimal point is always off.
    SELECTONE data FROM
        [0] : out = Xzero;    [1] : out = Xone;
        [2] : out = Xtwo;    [3] : out = Xthree;
        [4] : out = Xfour;    [5] : out = Xfive;
        [6] : out = Xsix;     [7] : out = Xseven;
        [8] : out = Xeight;   [9] : out = Xnine;
        [10] : out = XA;      [11] : out = XB;
        [12] : out = XC;      [13] : out = XD;
        [14] : out = XE;      [15] : out = XF;
    ENDSELECTONE;
    a = out<6>;              b = out<5>;      c = out<4>;      d = out<3>;
    e = out<2>;              f = out<1>;      g = out<0>;
ENDROUTINE;
ENDMODEL;

```

Figure A.4: File 7seg.bds.

```

        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_EDGE BOTTOM
        TERM_RELATIVE_POSITION 0.1
        TERM_RELATIVE_POSITION_STEP 0.2
FORMAL_TERMINAL data<3>
FORMAL_TERMINAL data<2>
FORMAL_TERMINAL data<1>
FORMAL_TERMINAL data<0>
        DIRECTION OUTPUT
        TERM_EDGE TOP
        TERM_RELATIVE_POSITION 0.1
        TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL a
FORMAL_TERMINAL b
FORMAL_TERMINAL c
FORMAL_TERMINAL d
FORMAL_TERMINAL e
FORMAL_TERMINAL f
FORMAL_TERMINAL g
FORMAL_TERMINAL dp

```

Figure A.5: The file 7seg.pads specifies the desired position of the terminals of the SSDD.

places visited, or on all the nodes visited (i.e., both transitions and places), with the graph traversed either backward or forward, going as many levels deep as you want. Try for example:

```

vov_sh -H -t2 -h 7seg:wolfe # transitions, 2 levels back
vov_sh -H -p2 -h 7seg:wolfe # places, 2 levels back
vov_sh -H -n20 -h 7seg:wolfe # nodes, 20 levels back
vov_sh -H -tpn400 -h 7seg:wolfe # everything, many many levels back
vov_sh -H +t3 -h 7seg.bds # transitions, 3 levels forward

```

A.2.7 Validity of nodes

Each node in the trace, whether it is a place or a transition, has a *status*. Normally the status is either `VALID` or `NOT VALID`, but there are other possibilities described in the second part. A `VALID` status means that the node is “good.” If the node is a transition, the transition has successfully completed. If the node is a place, it is up to date and consistent with all the other places it depends upon. If a transition is `NOT VALID`, it has not been run successfully yet, or it should be run again, probably because one of its inputs has been modified since the last time the transition was executed. If a place is `NOT VALID`, it is the output of a transition which is also `NOT VALID`.

The status of the nodes is managed by VOV. It is possible for the user to change the status of a node, but it is best that you do not do it in this exercise.

```

!! Make some vectors for convenience.
!!
mv OUT a b c d e f g
mv IN  data<3:0>

!! Macro with one parameter.
macro test
    set #c = #c + 1
    set IN #c          !! Set the input.
    ev                 !! Evaluate circuit.
    show IN OUT        !! Show signals.
$end
macro test4
    test
    test
    test
    test
$end
!! Now try all possible inputs.
set #c = 1111          !! So we start simulation from 0
test4
test4
test4
test4

quit !! This quit is a clean way to complete a script.

```

Figure A.6: File 7seg.musa.

```

$ vov_sh -h 7seg:wolfe

.../vovtutorial/andrea/7seg:wolfe:contents
NOT VALID  Fri Aug 24 14:56:56 1990(fornax)

NOT VALID vov_wolfe -f -r 2 -o 7seg:wolfe 7seg:padp
NOT VALID vov_padplace -D 7seg:pads -o 7seg:padp 7seg:logic

```

Figure A.7: The history of 7seg:wolfe.

A.2.8 Automatic retracing

When all of your files are ready, you can ask VOV to run all the tools for you, that is to retrace your design. It is always a good idea to begin slowly. In this case, try retracing one step at a time. The first transition to be executed should be `vov_bdsyn 7seg.bds` which outputs `7seg.blif`. If you just want to regenerate `7seg.blif` you'll say:

```
vov_sh -r 7seg.blif
```

The server determines what needs to be done to bring `7seg.blif` up to date (in our case, it will simply run `vov_bdsyn`).

VOV estimates the time it should take to do the retracing by adding up the estimated duration of each transition which is scheduled to be retraced, with the assumption that each transition is going to require as much time as when it was executed last. This estimation is sometimes wrong, but it tends to become accurate as the design data becomes more stable.

There might be some syntax errors in your BDS file. Read carefully the output you get from VOV. If there are errors, try to fix them, and do the retracing again.

If you are more confident in the correctness of your data, you could say, for example:

```
vov_sh -r 7seg:wolfe    # All the way down to 7seg:wolfe
vov_sh -k 7seg.bds      # Anything that depends on 7seg.bds
vov_sh -k 7seg.pads     # Anything that depends on 7seg.pads
vov_sh -R               # Retrace all (won't work, but try anyway)
vov_sh -AR              # Retrace all (this works)
```

Please note that some of these retracings will fail, as explained later in section A.2.11.

It is OK to ask for a global retrace (option `-R`) as long as you work alone. If you are part of a team, retracing everything would also retrace parts of the trace which “belong” to your teammates. Although this is not dangerous, it might result in a waste of CPU cycles.

If there are mistakes in your files, you will probably have to iterate this process a few times: modify files, retrace, modify, retrace.

A.2.9 Review what has happened

Use `vov_sh -n` to get a dialog that allows you to examine the most recent events related to the evolution of your design trace. This is especially useful to find out what, if anything, went wrong.

All the important events in the trace are recorded in a queue until a designer has inspected them. The inspection of the event queue gives you quick access to data and transitions, by a simple click of the `Edit` button. Please spend some time to experiment with this dialog (Figure A.8), because it is very useful. Try the `Previous` and the `Next` button, and see how they interact with the “FILTER” flags. Also notice how the horizontal scroll-bar on the top of the dialog gives you fine control on the event displayed on the lower window.

You can also interact with VOV using `vov_sh -i`, which brings up the dialog shown in Figure A.9. You are encouraged to experiment with this dialog, because you will find it useful.

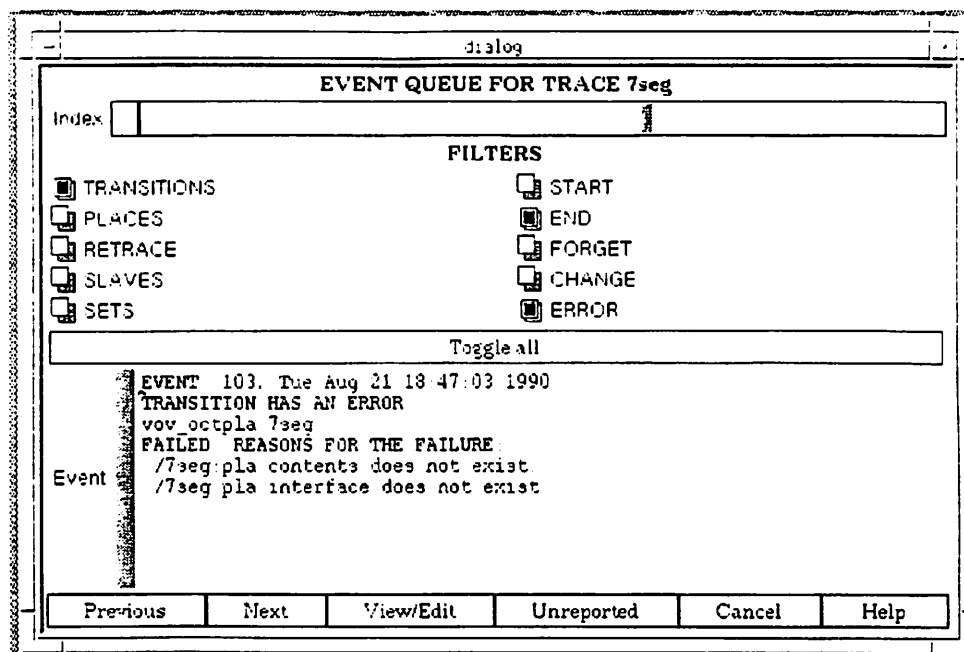
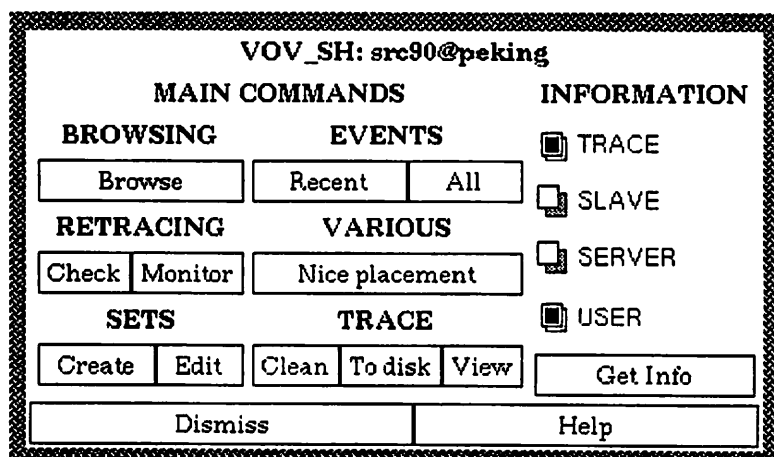


Figure A.8: The dialog to browse the event queue.

Figure A.9: Use `vov.sh -i` to get this control panel.

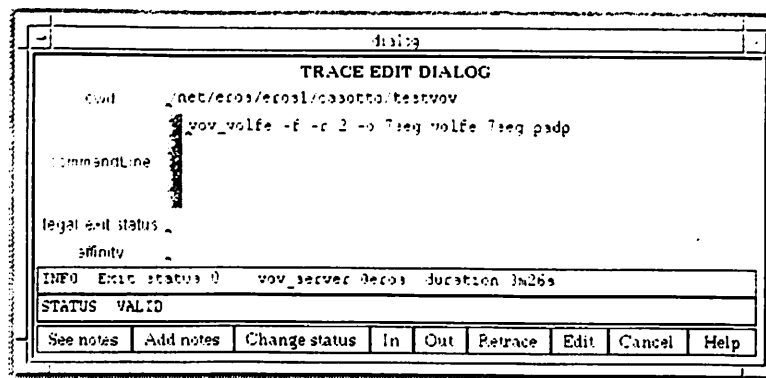


Figure A.10: The dialog to edit a transition.

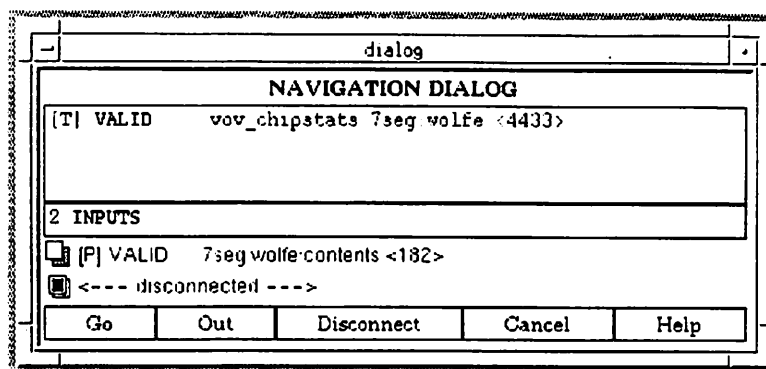


Figure A.11: This dialog shows the inputs of a node and allows the user to navigate the trace or to disconnect nodes.

A.2.10 Possible problems

It is not possible to list all the problems that can arise during your exercise. There can be syntax errors in some of the files, which can be resolved by studying the output of the tool which has detected the error.

Sometimes VOV will not retrace a transition because one or more of its inputs is not valid. This can happen for several reasons:

- VOV has just realized that an input has become invalid. Just ask for another retrace.
- The invalid input is bogus, it is never going to become valid. This is a situation of deadlock from which VOV cannot get out without your help. You can either repeat the transition by hand (type it again from your shell), or you can decide to remove the input from the dependency list. From the event-queue dialog locate the event describing the failed retracing of the transition and click on **Edit**. You get a dialog describing the transition (Figure A.10). Click on **In** to look at the inputs of the transition (Figure A.11). Select the offending bogus input (one at a time) and click on **Disconnect**. This removes the place from the input list for the transition thus effectively changing the criterion for the re-execution of the transition.

A.2.11 Substitution a transition

The example trace is not going to work with the data described in this tutorial. The reason is rather obscure for a novice: you should run `vov_bdsyn` with the `-o` option, which will omit the trailing `<0>` for 1-bit variables — what is expected by the files `7seg.pads` and `7seg.musa`. Thus, instead of the `vov_bdsyn 7seg.bds`, as suggested to you by the assistant, you should use

```
vov_bdsyn -o 7seg.bds
```

How do you get rid of the old transition and add the new one instead? Just run the transition by hand. Since you are effectively suggesting a new way to produce the file `7seg.blif`, VOV will ask you to confirm what you are doing. This is a case of “output conflict” detected by VOV; a place that is already the output of a transition is being declared as the output of another transition, i.e. the place is being “defined” again. Since the definition of a place must be unique, either the new transition is a mistake, or the old one must be removed (forgotten): you must decide.

For such a small change in a command line, you could have also edited the transition directly, using the event queue to find an event that mentions the transition and clicking on `Edit`.

A.2.12 Check the results

When the retracing has completed successfully, somewhere you have obtained the layout of the SSDD, in an OCT facet called `7seg:wolfe`. You can use VEM to take a look at it. The easiest way is to scan the event queue until you find the event describing the change of `7seg:wolfe`, click on `Edit`, double-click on the VEM command displayed in the edit dialog, and then deposit the command in the VEM console by clicking the middle button of the mouse (easier done than said.)

In a similar way you should find the event relative to a change in the output of `vov_chipstats` and take a look at the file. How large is the layout? How many nets are there? How many cells (instances)?

A.2.13 Try something new

Until now, you have followed a prepared trace, and it is finally time to start breaking new ground. Suppose that, by reading the Octtools manuals, you find out that `vov_musa` offers a nice graphical way to simulate seven segment displays, and you become convinced that this graphics would be the best way to test the correctness of the SSDD.

What is needed is a way to connect together your SSDD with a model of a seven segment display. You need a new circuit with one *instance* of a driver and one *instance* of a seven segment display. The *terminals* of the SSDD should be appropriately connected to the terminals of the display by means of *nets*. The new circuit should also have some *formal terminals* to talk to the outside world. All this can be done with the BDNET language, which is very useful to write netlists.

In the manuals you discover the syntax for BDNET and write the file shown in Figure A.12, which actually uses two displays, a red one and a yellow one, just for the fun of it.

Type:

```
vov_bdnet 7seg.bdnet
```

```

MODEL 7seg:symbolic;
TECHNOLOGY scmos; !! Required property.
VIEWTYPE SYMBOLIC; !! Required property.

!! Terminals of this circuit.
INPUT data<7:0>; SUPPLY Vdd; GROUND Gnd;

INSTANCE 7seg:wolfe NAME = HIDEDEC
data<3:0> : data<7:4>;
a : hiA; b : hiB;
c : hiC; d : hiD;
e : hiE; f : hiF;
g : hiG; dp : UNCONNECTED;
wolfe_Vdd : Vdd; wolfe_Gnd : Gnd;

INSTANCE 7seg:wolfe NAME = LODEC
data<3:0> : data<3:0>;
a : loA; b : loB;
c : loC; d : loD;
e : loE; f : loF;
g : loG; dp : UNCONNECTED;
wolfe_Vdd : Vdd; wolfe_Gnd : Gnd;

INSTANCE "~octtools/lib/musa/redseg":physical [50,0] NAME = HISEG;
COMMON : Gnd;
A : hiA; B : hiB;
C : hiC; D : hiD;
E : hiE; F : hiF;
G : hiG; DP : Gnd;

INSTANCE "~octtools/lib/musa/yellowseg":physical [200,0] NAME = LOSEG;
COMMON : Gnd;
A : loA; B : loB;
C : loC; D : loD;
E : loE; F : loF;
G : loG; DP : Gnd ;

ENDMODEL;

```

Figure A.12: The file 7seg.bdnet describes a netlist of two decoders each connected to a display.

```

mv DATA data<7:0>
macro test
    set #c = #c + 1
    set DATA #c
    ev
    sleep 200
$end
macro test4
    test
    test
    test
    test
$end
macro test16
    test4
    test4
    test4
    test4
$end
macro test64
    test16
    test16
    test16
    test16
$end
set #c = 11111111
test64
test64
test64
test64
sleep 2000    !! Rest for a while.
quit

```

Figure A.13: The simulation script to test the circuit with two seven segment displays.

The tool `vov_bdnet` connects to the server and registers itself, with its inputs and outputs. From now on, VOV knows that you want to run `vov_bdnet` on the file `7seg.bdnet`. If you change the file, you can ask for a retrace using, for example,

```
vov_sh -k 7seg.bdnet
```

When your netlist is correct, you can run `vov_musa` on the new circuit. Of course you need a completely different script to simulate this circuit, so you write the file shown in Figure A.13 which you call `double7seg.musa`.

Then you run the simulator:

```
vov_musa -i double7seg.musa 7seg:symbolic
```

This transition is interactive. It does not make sense to run it if there is nobody watching the graphical output to decide if it is correct or not. If, for any reason, you ask VOV to retrace the

transition, VOV will obediently do it, but you will lose all the graphical display. If you want more control on retracing of interactive transitions, please read the second part of this tutorial.

A.2.14 Suspension or end of the exercise

If you want to end or suspend the tutorial, it is better that you kill the server. This is optional, more than anything it is a courtesy to your colleagues that might want to use the machine. For a real design, it will be better to keep the server running continuously, until the project is completed.

```
vov_sh -AK    ## Kill the server gently.
```

The server saves the trace one last time in the directory `~/minivov` (with the name `<projectname>.page.0:symbolic`) and exits. You can obtain a hardcopy of your trace using the program `vov_hardcopy`:

```
vov_hardcopy <projectname> [printername]
```

You can use `vov_mini` again to restart the server.

A.3 TUTORIAL: Second part

The program `vov_mini`, used in the tutorial, is a convenient way to get VOV started. It is called “mini” only because it is so easy to start, but in reality it gives you the full capabilities of VOV, the only limitation being that only one slave is connected to your server.

A.3.1 What does `vov_mini` really do

The steps involved in `vov_mini` are:

- Create, if it does not exist already, the directory `~/minivov`.
- Prepare a “slaves” file, containing only one entry for a slave to be run on the local machine. The file is called `<projectname>.vov.slaves`.
- Start the program `vov_server` with the options `-B -S`. The option `-B` makes the server run in batch mode, which means that the server automatically recovers in case of errors (e.g. errors in the communication protocol). The `-S` option makes the server start the slaves described in the “slaves” file.
- Create a new alias in the file `~/ .vovprojects`. The name of the alias is the `prprojectname`. The alias can be used to set the two environment variables used by the VOV clients to locate the server in the among the machines in the network. The two environment variables are: `VOV_HOST_NAME` and `VOV_PROJECT_NAME`.

If you want to work with your server from another window (possibly in another machine), you simply have to invoke this alias and then all clients will connect to the server.

You can have more control on VOV if you perform each of these steps by hand. For example, you can decide the working directory of the server, or you can have a dozen slaves to work for you.

```

# Slave list for project: BRIC
# Lines beginning with '#' are comments.
# Each line MUST contain:
# HOST_NAME: name of the host where you want the slave to run.
# COEFF:      An integer number used to balance the relative
#             power of the slaves.
#             COEFF = 1 means: Use actual power of the slave
#             COEFF = 4 means: Divide actual power of slave by 4.
# MAX_LOAD:   Maximum load allowed on the host in order to dispatch a
#             job to the slave. If the load on the host is greater
#             than MAX_LOAD, no job will be dispatched to the slave.
#             Optionally, you can add a list of resources provided by each slave
#             within a pair of parentheses. Example: hostname 1 4 ( vax verilog )
# IMPORTANT: leave blanks before and after the parentheses.
#
# HOST_NAME      COEFF    MAX_LOAD (AFFINITY)
# aahz           2        0.9
# chumly         2        0.9
# eros           1        8 ( bigvax vax )
# sequent        1        0.8

```

Figure A.14: Example of a “slaves” file.

A.3.2 Add many slaves to your server

A slave is a special client, which simply waits to be assigned something to do. When the server has something to be retraced, it chooses the best possible slave for that job. It is convenient to have one slave running on all the machines on which you have accounts (of course, all the slaves must be able to get to the design data, possibly through NFS).

The best way to connect several slaves is to prepare a “slaves” file, such as the one shown in Figure A.14, which should be called `<projectname>.vov.slaves` and should reside in the current working directory of the server (`~/minivov` if you use `vov_mini`).

All slaves are ranked by “power,” a number which takes into account the speed of the CPU and the load on the machine. The coefficient `COEFF` is a politeness coefficient: it should be a positive integer. It is used as a divisor in computing the power of a slave. Thus, a slave with a `COEFF=2` has half the power of an identical slave with `COEFF=1`. If you start a slave on a machine normally used by somebody else, it is a good practice to have VOV use that slave only if all other slaves running on your machines are already busy. This behavior can be controlled by `COEFF`. In most cases, however, it is always a good idea to leave `COEFF=1`. The number in the `MAX_LOAD` field specifies that the slave should not accept jobs if the load on the machine is greater than the number. The `AFFINITY` is an optional field. Be careful if you use it: you need to put spaces between the parenthesis and anything else. See section A.3.9 for more details.

When the file is ready, you should type:

```
vov_start_slaves <hostname> <projectname>
```

to get all the slaves started. `<hostname>` is the name of the host in which the server is running. You can also have the server start the slaves automatically by using the “-S” option in `vov_server`.

The status of the slaves can be checked with `vov_sh -m`, or with the “Monitor” button in `vov_sh -i`.

You should not have more than one slave running on each machine, or you will only overload that machine. The server can have at most 64 clients connected at the same time. It is better not to connect more than about 20 slaves, or you will run out of connection points.

Slaves can be killed at any time with an ordinary `ctrl-c` or with `kill -9`.

You can also start slaves on the fly. Just log onto a machine, set the correct environment variables and say:

```
vov_slave
```

When you no longer need this slave, just kill it with `ctrl-c`.

Each slave makes available a set of resources, identified by a set of words. By default, a slave provides two resources, the name of the host on which the slave is running and the machine type. For example a slave running on the VAX called “formax” offers the resources “vax formax”. You can override the set of resources offered by the slave with the `-A` option. A special slave which uses the `-A` option is

```
vov_interactive
```

which offers the resource “interactive”, which makes the slave eligible to accept transitions whose affinity is “interactive”.

A.3.3 Start the server

You must decide which host you want the server to run on. A good host is one that has NFS access to all files used for the project. You can run several servers on the same host, as long as they are given different project names. Typically, there is one server per project.

To restart the server use:

```
cd <server_working_directory>
vov_server -S -B <projectname> >& outfile &
```

If you are starting a new project, you have to use the `-C` option as well. Do not use this option when you RESTART the server, or you will lose the trace.

The server has been designed to run continuously for the whole duration of the project. You should not kill the server if you log out.

A.3.4 Clients

How do clients find which server to connect to? They use two environment variables: `VOV_HOST_NAME` and `VOV_PROJECT_NAME`. These variables are set with the `setenv` command:

```
setenv VOV_HOST_NAME <hostname>
setenv VOV_PROJECT_NAME <projectname>
```

It is convenient to create an alias to set those environment variables. All this is done for you by `vov_setup`, which updates or creates the file `~/.vovprojects`. For example, if you have a project called BRIC with the server running on eras, you would say:


```
vov_setup eros BRIC
```

and you would then have a new alias

```
alias BRIC 'setenv VOV_HOST_NAME eros; setenv VOV_PROJECT_NAME BRIC'
```

In this way, you just have to type BRIC to set the variables.

Another environment variable used by the clients is VOV_TRACE_ONLY which, if set to TRUE, tells the client that we are only interested in the trace left by the execution, and not in the execution itself. This is useful when you want to build a large trace involving many time consuming transactions. Setting VOV_TRACE_ONLY, you have a quick way to produce a large trace and then you can let VOV do the retracing for you.

A.3.5 The event queue and the journal

The server keeps a record of everything that happens in the trace. All events are recorded in a journal file which resides in the working directory of the server. The journal is called `journal.<project>`. The journal is of little practical utility, and it is mostly used to do detective work to find out why something strange and unexpected happened.

A.3.6 The event queue

To take a look at the latest events, you can also use `vov_sh -n`, where “-n” should remind you of the word “notify.” You should already be familiar with the dialog that allows you to browse the event queue. It offers you the quickest way to the objects in the trace. For example, if you want to change the command line of a transition, without retyping everything, you can find an event that mentions the transition, click the “Edit” button, and do the changes you want to do in the transition editing dialog that has popped up.

A.3.7 The trace

The trace is stored in an OCT symbolic facet, called `BRIC.page.0:symbolic`. It might be interesting to look at the trace with `vem`. The most current version of the facet is always in the memory of the server, which saves the trace occasionally (about once every hour). If you want to see the latest version of the trace you can force the server to save the trace onto disk with `vov_sh -w`. Also you can use `vov_rpc` to browse through the trace.

`vov_sh` is the main tool to get information from the server. In particular:

```
vov_sh -I           : Get information on what is going on.
vov_sh -h place    : Gives you the history of a place (file or facet)
vov_sh -w          : Force the server to write the trace to disk.
vov_sh -t place    : Toggle the status of the place.
vov_sh -AR         : Retrace everything that has to be retraced.
vov_sh -r place    : Retrace what is needed to make ``place`` consistent.
vov_sh -k place    : Kick ``place``, i.e. retrace all its dependents.
vov_sh -n          : Be notified about the latest events.
vov_sh -i          : Start the interactive interface for vov_sh.
```

You will most frequently use the interactive interface of `vov_sh`.

A.3.8 Annotations

Adding annotations to the trace is like adding comments to a program. Everyone agrees that it is important, but no one really does it because it is a hassle. In case you are a well disciplined designer and want to carefully document what you are doing, for your own benefit or for the benefit of your teammates, VOV gives you the possibility to add **annotations** to any element in the design trace. An annotation is a piece of text attached to the object. Use annotations to explain why you did something, or the meaning of a place. Whenever you are editing or viewing an object, please notice in the lower left corner of the dialog the buttons labeled "See notes" and "Add notes". The "See notes" button appears only if the object already has annotations attached to it, the other button is always there. If you click either button you get a dialog to browse, add or forget the annotations for the object.

A.3.9 Affinity of transition, interactive transitions

In VOV it is possible to connect to the server several slaves, each running on a different machine in the LAN. The slaves provide VOV with CPU cycles to perform the necessary retracing. Two slaves are **equivalent** if any transition performed on one slave would give the same results if executed on the other slave.

It would be ideal if all slaves were equivalent: the dispatching of transitions could be based only on the greedy strategy that the best slave gets the transition with longest expected duration. In practice, different machines offer different resources and slaves can lose equivalence for several reasons:

architecture: the output of a compilation depends on the architecture of the machine;

hardware resources: some transaction could be very time consuming, or require a large amount of memory, so that it should only be executed on the large machines in the network, rather than on the smaller workstations;

software resources: some commercial software such as "verilog" can run only on the machines which have been licensed.

One unsatisfactory solution is to limit VOV to the largest subset of equivalent slaves. This might not work because in general, a subset of equivalent slave does not completely cover the set of resources needed to complete a design, or the subset could be too small for the design.

VOV deals with the problem of non-equivalent slaves by considering the **affinity** of each transition. If a transition, for licensing reasons, can only run on the machines called "pinocchio" and "geppetto," its affinity is "pinocchio geppetto". If a transition can only run on vax'es, its affinity would be "vax". For the special case of a transition that can run on any machine, the affinity would be an empty string. By default, the affinity list of a transition contains only the machine type.

In a similar fashion, each slave has an associated list of resources, which defaults to a list containing the machine name and the machine type. Thus, the resource list of a slave running on the VAX "pinocchio" would be "pinocchio vax".

Both the affinity list of a transition and the resource list of a slave can be overridden by the user.

A *match* between an affinity list and a resource list exists if either string is empty or if the two lists have one word in common. When dispatching a transition to one of the slaves, VOV scans the slaves in decreasing order of power, and chooses the first idle slave whose resource list matches the affinity list of the transition.

A.3.10 Graphical interface using vem/RPC

A simple RPC application is available to browse the trace. Move to your project directory before you type: `vov_rpc BRIC` This starts `vem` and an RPC application called `vovRpc`.

The color coding used has the following meaning:

```
pink (MET2)      : valid trace
blue (MET1)      : invalid trace
red (POLY)       : active tracing (a transaction is currently executing)
orange (NWEL)    : active retracing.
```

The places in the trace use different icons to ease the understanding of the graph. Here is the key to interpret the icons:

```
circle : Data file (often an ASCII file)
octagon : OCT facet
X       : Executable
S       : Exist status
O       : Command line option
```

A.3.11 Status of the trace

Each node in the trace has a "status," which can be one of the following:

VALID: The node is "good." If the node is a transition, the transition has successfully completed. If the node is a place, it is up to date and consistent with all the other places it depends upon.

NOT VALID: If the node is a transition, it has not been run successfully yet, or it should be run again, probably because one of its inputs has been modified since the last time the transition was executed. If the node is a place, it is the output of a transition which is also NOT VALID.

TRACING: If the node is a transition, it is currently being executed for the first time. All the outputs of such transition are also in the same status.

RETRACING: If the node is a transition, it is currently being retraced. All the outputs of such transition are also in the same status.

DEAD: The dead parts of the trace are ignored by the server. A node can become DEAD only if a designer decides so. Dead nodes are kept for documentation purposes, or to prevent the server from retracing them.

MISSING: This status is used for places which suddenly disappear.

WEIRD: You might occasionally see this as a possible status, but in reality it is not used, and it will soon disappear.

A.3.12 Protection

Currently VOV is an open system, which ignores protections. All the data generated by VOV is not protected. Although this is only a temporary situation, it should not be a problem.

A.3.13 Sets

You can create sets of nodes. You can do this from the `vov_sh -i` menu. When you create a set you must specify a name for the set, and a **selection rule**. The selection rule is complicated: you can select nodes by type (PLACE or TRANSITION), by status (VALID or NOT VALID or DEAD or any of the others), by type of place (UNIX FILE, OCT FACET, EXECUTABLE, etc.) or by regular expression matching. You can also select nodes with no inputs, with no outputs, or isolated nodes, those with no inputs and no outputs.

The regular expression matching routines use the “emacs” syntax. The dot `.` stands for any character, the star `*` stands for an arbitrary repetition of the previous expression. Thus the regular expression `.*` matches every string (any character repeated any number of times). If you want to make a set of all the files whose name begin with `.std.` you should specify the regular expression `.*\.std\..*` in which `\.` matches the dot. For a more complete description of the syntax, ask emacs.

Why should you want to make sets? A number of reasons, the most important being documentation and because you can use sets to delete (forget) nodes from the trace.

A.3.14 Forgetting nodes

VOV makes it purposely hard to eliminate nodes from the trace. The simplest way to tell VOV to ignore parts of the trace is to change the status of the uninteresting nodes to DEAD. Do this with the “Change status” button you can see in many dialogs. But if you are positively sure that you want VOV to forget about something, here is what you do. First you must start `vov_sh -i` to get the menu that allows to create and edit sets. Then you create a set containing the nodes you want to forget. Then you edit the set and ask for a list of its elements. Then you select the nodes in the set that you want to forget and hit the “Forget” button.

Sometimes, before you forget a place, you might want to delete it from disk and throw it in the trash. Thus you can select some nodes and hit “Trash”. Actually VOV does not remove the files from disk, it just puts them in another directory called `VOV_GARBAGE_CAN` in the working directory of the server.

A.3.15 Moving stuff around the file system

Would you like to change the path name of a place in the trace? Use the “Mvlib” button in the `vov_sh -Ai` dialog. Alternatively you can use the `-O/-N` pair of options in `vov_sh`. Example. Suppose you have built a pla in a directory `/users/joe/mypla` and have decided to move all the data in another directory, for example by doing:

```
cd /users/joe
mv mypla reallygoodpla
```

Appendix B

Quick Tool Overview

The tools listed below in alphabetical order are mentioned in the examples.

bdnet is used to create OCT netlists starting from a textual description.

bdsyn is a translator of logic equations, from the compact BDS format into the expanded BLIF format that is suitable input for logic optimizers such as **misII**.

chipstats measures area and net-length of a chip.

misII is a manipulator of logic equations. It is mostly used to perform logic optimization and occasionally to do format translation, for example between BLIF and OCT.

mosaico is a collection of tools for routing of macro-cell chips. The main tools in **Mosaico** are: **atlas**, for channel definition, the global router **cds**, the detailed router pair consisting of **cprep** and **spider**, the via minimizer **mizer**, the hierarchy flattener **octflatten**, the symbolic compactor and spacer **sparscs**.

musa is the multi-level logic simulator of the *Octtools*.

padplace is a multi-purpose tool used to handle the formal terminals of OCT facets, with specialized routines for pads.

sparscs is the compactor and spacer for symbolic layout.

vulcan creates abstractions of layout, by computing simplified protection frames for each layer in the layout.

wolfe places and routes standard-cell circuits.

Now VOV is surely confused. It knew about some files in the old directory and now it is unable to find them. So you now just tell VOV what happened:

```
vov_sh -O /users/joe/mypla -N /users/joe/reallygoodpla
```

A.3.16 Handy utilities

vov_cleanup : get a report on what files are useful and what can be removed.

vov_setup : create a new alias to set the VOV environment variables.

vov_start_slaves : start slaves described in the slaves file.

vov_kill_slaves : kill slaves described in the slaves file.

- [14] C.B. Shung, et.al. An Integrated CAD System for Algorithm-Specific IC Design. *IEEE Transactions on Computer Aided Design*, 1990. Accepted for publication.
- [15] TziCher Chiueh, Randy Katz, and Valerie King. A history model for managing the VLSI design process. In *ICCAD*, 1990.
- [16] G.C. Clark and R.E. Zippel. Schema: An architecture for knowledge based CAD. In ????, 1985.
- [17] James Daniell and Stephen W. Director. An object oriented approach to CAD tool control. In *26th Design Automation Conference*, pages 197–202, June 1989.
- [18] Allen M. Dewey and Stephen W. Director. Yoda: A framework for the conceptual design VLSI design systems. In *Proc. of ICCAD*, November 1989.
- [19] Susan A. Ellis. A symbolic layout language and a database for an integrated vlsi design system. Technical report, Electronics Research Lab, University of California, 1981.
- [20] D. D. Gajski and R. H. Kuhn. New VLSI tools. *IEEE Computer Magazine*, 16, December 1983.
- [21] Daniel D. Gajski and Donald E. Thomas. *Introduction to Silicon Compilation*, chapter 1. Addison-Wesley Publishing Company, 1988.
- [22] D.D. Gajsky, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A second opinion on data flow machines and languages. *Computer*, 15(2), Feb 1982.
- [23] G. Genrich. *Predicate/Transition Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer Publishing Company, 1987.
- [24] Kyle Goldman and Ted Stout. A design automation environment. *VLSI Systems Design*, pages 46–49, June 1988.
- [25] I. Goldstein and D. Bobrow. A layered approach to software design. In *Interactive Programming Environments*. McGraw-Hill, New York, 1987. D. Barstow and H. Shrobe and E. Sandwell.
- [26] Frank G. Halasz. Reflections on Notecards: seven issues for the next generation hypermedia systems. *Communications of the ACM*, 31(7), July 1988.
- [27] D. Harrison, P. Moore, Rick L. Spickelmier, and A. R. Newton. Data management and graphics editing in the Berkeley Design Environment. In *Proc. ICCAD*, pages 24–27, 1986.
- [28] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes. Electronic CAD Frameworks. *Proceedings of the IEEE*, pages 393–417, Feb 1990.
- [29] Paul Heckel. *The Elements of Friendly Software Design*. Warner Books, 1984.
- [30] Alberto Di Janni. A Monitor for complex CAD systems. In *Proc. 23rd Design Automation Conference*, pages 145–151, 1986.

Bibliography

- [1] William B. Ackerman. Data flow languages. *Computer*, 15(2), Feb 1982.
- [2] T. A. Agerwala. A complete model for representing the coordination of asynchronous processes. Technical report, Hopkins Computer Research Report No. 32, July 1974.
- [3] Wayne Allen, Ken Fiduk, and Doug Rosenthal. Distributed methodology management for design in-the-large. In *ICCAD*, 1990.
- [4] François Bancilhon, Won Kim, and Henry F. Korth. A model of CAD transactions. In *Proceedings of VLDB 85*, Stockholm, 1985.
- [5] D.S. Batory and Won Kim. Support for versions of VLSI CAD objects. Technical report, MCC, Austin TX, 1985.
- [6] Felix Bretschneider and Helmut Lager. Knowledge based design flow management. In *Proc of Conference on AI, Simulation and planning in High Autonomous Systems*, Tucson, Arizona, 26/27 March 1990.
- [7] Felix Bretschneider, Christa Kopf, Helmut Lager, Arding Hsu, and Elizabeth Wei. Knowledge based design flow management. In *ICCAD*, 1990.
- [8] Jean Brouwers and Moshe Gray. Integrating the electronic design process. *VLSI Systems Design*, June 1988.
- [9] Misha R. Buric and Thomas G. Matheson. Silicon compilation environments. In *Proc. of Custom Integrated Circuits Conference*, pages 208–212, 1985.
- [10] Michael L. Bushnell. *ULYSSES – An Expert-System Based VLSI Design Environment*. PhD thesis, CMU, research report CMUCAD-87-15, May 1987.
- [11] Michael L. Bushnell and S. W. Director. VLSI CAD tool integration using the Ulysses environment. In *23rd Design Automation Conference*, pages 55–61, 1986.
- [12] Andrea Casotto, Chuck Kring, and Randy Katz. Using the Oct-tools in a VLSI design course. In *1989 VLSI Education Conference & Exposition*, pages 105–117, July 1989.
- [13] Andrea Casotto, A. Richard Newton, and Alberto Sangiovanni-Vincentelli. Design management based on design traces. In *27th Design Automation Conference*, Orlando, FLA, June 1990.

- [48] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [49] P. van der Hamer and M.A. Treffers. A data flow architecture for CAD frameworks. In *Proc. of ICCAD*, pages 482-485, 1990.
- [50] P. van der Wolf, P. Bingley, and P. Dewilde. On the architecture of a CAD framework: the NELSI approach. In *Proc. EDAC 90*, 1990.
- [51] I. Widya, T.G.R van der Leuken, and P. van der Wolf. Concurrency control in a VLSI design database. In *25th Design Automation Conference*, 1988.
- [52] Gerhard Zimmerman. PLAYOUT - a hierarchical design system. In *IFIP*, 1989.
- [53] Michael D. Zisman. Use of production systems for modeling asynchronous concurrent processes. Academic Press Inc., in *PATTERN-DIRECTED INFERENCE SYSTEMS*, 1978. University of Pennsylvania.

- [31] Randy H. Katz, Rajiv Bhateja, Ellis E-Li Chang, David Gedyce, and Vony Trijanto. Design version management. *IEEE Design & Test*, pages 12–21, Feb 1987.
- [32] Ken H. Keller. An electronic circuit cad framework. Technical report, M84/54, Electronics Research Lab, University of California, July 1984.
- [33] David W. Knapp. *A Planning Model of the Design Process*. PhD thesis, USC, tech. rep. CRI-87-06, Dec 1986.
- [34] Krzysztof Kozminski. Design control in MCNC's open architecture silicon implementation system OASIS. Technical report, MCNC, Technical Report TR89-54, Dec 1989.
- [35] Alison Lee. Use of history for user support. Technical Report CSRI-212, University of Toronto, Computer Systems Research Institute, May 1988.
- [36] Steven S. Leung, P. David Fisher, and Michael A. Shanblatt. A conceptual framework for ASIC design. *Proceedings of the IEEE*, pages 741–755, July 1988.
- [37] Willems W. G. H. M. A VLSI Design Manager based on State Management. Technical report, Master thesis, Delft University of Technology, September 1987.
- [38] Robin L. Steele (NCR Microelectronics). An expert system application in semicustom VLSI design. In *ACM/IEEE Design Automation Conference*, pages 679–686, 1987.
- [39] Toshiaki Miyazaki, Tamio Hoshino, and Makoto Endo. A CAD process scheduling technique. In *ICCAD*, 1990.
- [40] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [41] A.R. Newton, D. O. Pederson, A. L. Sangiovanni-Vincentelli, and C. H. Sequin. Design aids for VLSI: the Berkeley perspective. *IEEE Trans. Circuit and Systems*, CAS-28:666–680, July 1981.
- [42] John K. Ousterout, Andrew R. Cherenson, Frederick Douglass, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *COMPUTER*, pages 23–36, February 1988.
- [43] Interact advertisement, June 1990.
- [44] Richard Rubinstein and Harry Hersh. *The Human Factor*. Digital Press, 1984.
- [45] Carlo H. Sequin. Managing VLSI complexity: an outlook. *Proceedings of the IEEE*, 71(1), Jan 1983.
- [46] Ernst Siepmann. A data management interface as part of the framework of an integrated VLSI-design system. In *ICCAD*, 1989.
- [47] G.W. Sloof, P. Bingley, P. Dewilde, T.G.R. van Leuken, and P. van der Wolf. Design data management in a distributed hardware environment. In *Proc. EDAC 90*, 1990.

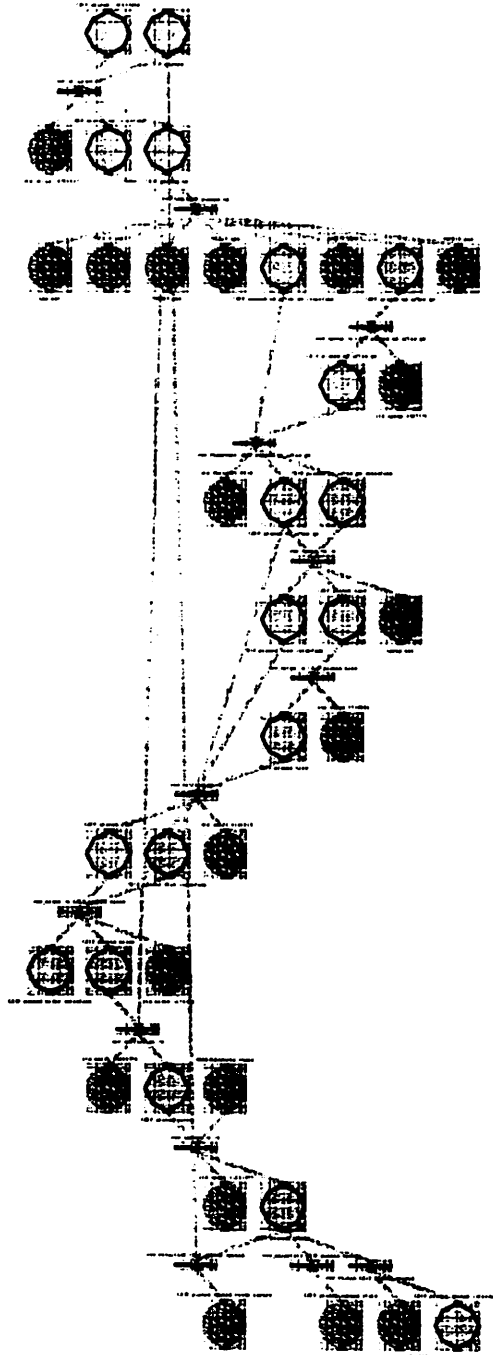


Figure 3.24: The Mosaico trace shows the sequence of tools required to route a macro-cell chip.

Chapter 4

Implementation

The development of a DMS is more an engineering problem than a scientific one. The solution derives from a balance of options and tradeoffs that cannot be abstractly validated on paper. The only experiment that can validate a DMS is to see how the system reacts to the complexity of the real world, a world in which real designers are determined to achieve a particular goal. This is an expensive experiment, because it requires the investment of a lot of energy into the development of the DMS. It is in fact necessary that the DMS is robust and reliable, powerful and lightweight, friendly and predictable, or the experiment will be corrupted by the designers' understandable unwillingness to cope with a sluggish and unreliable system.

The implementation of VOV has been a key element of this project, and is tightly connected to the conceptual development of the system. For example, the importance of the notion of affinity of transitions became clear when a prototype of VOV that did not have such notion was impossible to use because it could not dispatch some transitions to certain slaves. Once the notion of affinity was implemented, its implications on the firing rule became clearer, and its usefulness was extended to include the management of transitions competing for the same resource and of interactive transitions.

In the following sections, we overview some implementation issues encountered in the development of VOV. Section 4.1 develops some topics about the implementation of the objects in the trace and about the representation of the trace. Section 4.2 presents some some special topics such as safety and identification of traces. Some key features of the software implementation are highlighted in Section 4.3. The final section reports the raw performance of the current implementation of the system.

4.1 The design trace

4.1.1 Attributes of nodes

The most important attribute of a node is its status, which can take one of the following values: VALID, NOT VALID, DEAD, TRACING, RETRACING, MISSING.

VALID nodes are up-to-date and do not need retracing; they are either a primary input or the output of a successful transition. If a primary input changes, all of its dependent nodes become NOT VALID. NOT VALID nodes need to be retraced. DEAD nodes are ignored by the server; they are neither checked nor retraced. DEAD nodes are often used by novice designers who try some tools and then tell VOV to ignore what they have done. A DEAD node is useful for documentation purposes.

A currently executing transition and its outputs are in the TRACING status. The RETRACING status is reserved for transitions that are being retraced and for their outputs. A place can also be MISSING, when it is no longer on the disk. This is the normal status for temporary files that are created by some transition and then deleted by the designer.

Users can control the status of a node, and a number of rules determine the effect of a change in status of a node upon its dependent nodes. If a user invalidates a node, all of the dependent nodes also become NOT VALID (unless they are DEAD, in which case they stay DEAD). If a user forces a node to become VALID, its dependent nodes are not affected. Expert users can alter this default behavior and decide the status of entire subtraces. User cannot set the status of a node to either TRACING or RETRACING, because those values are reserved by the system.

4.1.2 Attributes of places and transitions

A **place** represents a piece of design data. In particular, VOV considers the following types of data: UNIX ASCII files, UNIX binaries, OCT facets, command line options, exit status of transitions, boolean conditions, measurements.

The *type* of a place determines the database that is managing the place and therefore it also determines the methods to operate on the place. For example, ASCII files are managed by UNIX and are manipulated with UNIX routines such as `write`, `read`, `stat`, `unlink`, while OCT facets are manipulated with the analogous procedures provided by OCT. Command line options, boolean places, exit status, and measures, are managed by VOV, which provides the methods to create, edit and delete such places.

Places have two other attributes: the *name*, which is a unique string used for identification of the place in its database, and the *timestamp*, which is the date in which the place was last modified. Timestamps for UNIX files are obtained directly from UNIX, but VOV must overcome two problems, caused by clock skews between different machines, and by the NFS caching mechanism. A file can be on a file system physically mounted on a different host from the one where the server is running. If the clocks on the two hosts are skewed the timestamp of the file must be adjusted to a reference clock, which is the one of the server.

If files are accessed through the Network File System protocol (NFS), one must consider that the protocol is not completely transparent, because some information is cached on the server side and caches are sometimes refreshed with a 30 to 40 second delay. It is therefore possible for a file to change and for the server to be unaware of the change until the caches are refreshed. This can cause a lot of confusion.

Consider a transition with a duration of just a couple of seconds. Suppose that the transition executes on a host different from the server's, and that the transition declares as output the file *File*. If, upon termination of the transition, the VOV server cannot see any change in the timestamp of *File*, because it is seeing the cached copy, it must conclude that the transition has failed, because it has not produced one of the outputs it had promised; *File* is marked as NOT VALID. 30 to 40 seconds later the server suddenly sees a change in *File*, and it becomes confused on what the status of *File* should be, so it sets it to VALID hoping for the best. To this date, no solution has been found to this NFS caching problem. The only clean solution is to eliminate NFS from the loop and have the server running on the same host where the data are stored (the clients can still be running on any host), but this defeats the stride for a really distributed system. It is hoped that some of other protocols developed for distributed file systems, such as the one implemented in the SPRITE [42] operating system, will soon become available on most UNIX platforms.

The attributes of a transition are the *command line*, the *working directory*, the *user name* who initiated the transition, the *host name* of the machine used for the transition, the *start date*, and the *finish date*. The *process id* is useful for job-control, because it permits VOV to stop the transition if it has been determined that its outcome is no longer of interest. The *exit status* of each process is also recorded and matched against a list of *legal exit status* to help determine whether the transition was successful. Finally, the *affinity* of the transition indicates if the transition requires special resources.

Some attributes of transitions are preserved during retracing, namely the original user name, the affinity, and the list of legal exit status.

4.1.3 Canonical names for files

Except for chains of places (Section 3.3), each place in the trace must have a unique name. In the case of UNIX files, this name should be meaningful for all processes involved in VOV, regardless of the host on which the process is executing. If all files are managed by one file server, then it is normally possible to use as name of the place the full path of the file. In the case in which the data is distributed across several file servers, the full path may point to different files if computed on different machines. The rule to generate a name meaningful for all machines depends of course on the particular way in which the file systems are mounted among the machines. For example, in the Berkeley CAD group, a file system `aaa` physically mounted on the machine `host1` can be accessed by another machine through the path `/net/host1/aaa`, using NFS. On `host1` itself, the path `/net/host1` exists and points to the root directory. The rule is therefore to add the prefix `/net/<hostname>` to a full path to obtain a file name that is valid network wide.

The mechanism of hard links and symbolic links allows UNIX to refer to the same file with two different names. While hard links are not considered, VOV repeatedly expands symbolic links and removes all occurrences of `“.”` and `“..”` in a path until a full path without symbolic links and dots is obtained. This path, possibly with a network prefix is the **canonical name** for the file, and it is the name used to identify the place that represents the file.

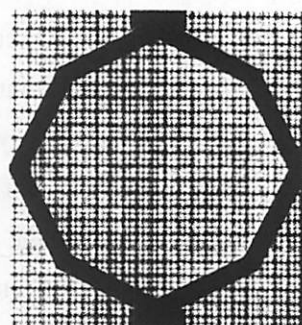
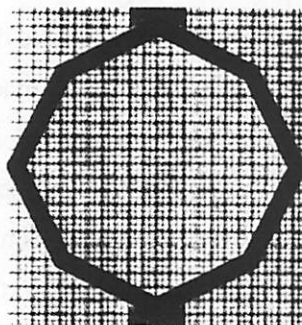
4.1.4 The representation of the trace

An early implementation of VOV used an internal C++ data structure to represent the trace, while another representation in OCT was used to provide persistency and the graphical user interface. That implementation failed because of difficulties in maintaining the consistency of the two representations. Using OCT also for the internal representation turned out to be a good choice because the code was greatly simplified.

The graphical representation of the trace is important for the designer, because it is a powerful way to communicate information about the flow of data. In order to exploit this potential, care has been taken to improve the readability of the graphical rendering of the trace, resulting in images such as those in Figure 4.1.

Each node in the trace is represented in OCT by an instance of some icon depending on the type of the node, as shown in Figures 3.1 and 3.2. The input/output relationship between nodes is represented by attachments among the instances: each node contains its outputs and is contained by its inputs. These attachments, however, have no graphical representation. Thus, for the benefit

decoder:padp:interface

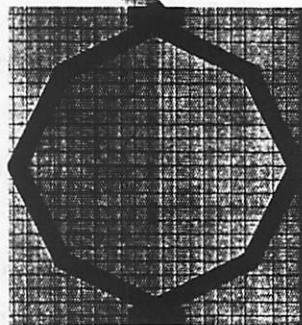
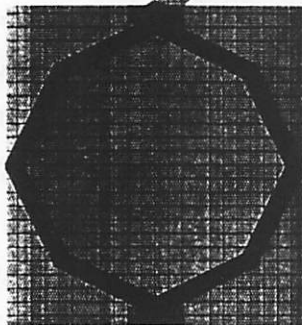


decoder:padp



vov wolfe f -o

decoder:wolfe:interface



decoder:wolfe

Figure 4.1: A small trace to highlight the features of the graphical representation.

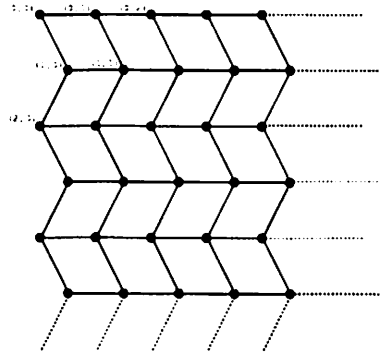


Figure 4.2: The grid used to place the nodes in the trace. The vertical coordinate of a node is determined by its *level*, the horizontal direction is chosen so that the total length of the arcs is minimized.

of the users, each dependency between two nodes is also represented by a straight colored path stretching from one node to the other. The color of the arc is the same as the color of its input node.

A color coding scheme has proved to be particularly effective to communicate information on the status of a design. For example, the designer can immediately see which nodes are VALID and which are not, because VALID nodes are pink, NOT VALID nodes are blue, DEAD nodes are dark green, TRACING nodes are red and RETRACING nodes are orange, MISSING nodes are brown.

The nodes are placed at the vertices of a grid that is derived from a semi-infinite rectangular grid by offsetting all the nodes on odd rows by one half of the row spacing, as shown in Figure 4.2. The origin of the grid is in the top left corner. The vertical coordinate of a node is determined by its *level*, with level zero corresponding to the top row. The level of a node represents its distance from the furthest primary input. More formally, the level $l(n)$ of node n is assigned in two steps. Step 1 assigns a level 1 to each primary input and to all other nodes it assigns the maximum level of their inputs increased by 1:

$$l(n) = \begin{cases} 1 & \text{if } I(n) = \emptyset \\ \max_{m \in I(n)} l(m) + 1 & \text{if } I(n) \neq \emptyset \end{cases}$$

Step 2 is purely cosmetic, in the sense that it tries to bring the primary inputs closer to the transitions that uses them first:

$$\forall n | I(n) = \emptyset \Rightarrow l(n) = \min_{m \in O(n)} l(m) - 1$$

During the evolution of the trace, a node may become disconnected, in which case its level is zero.

After levels have been assigned, the relative position of the nodes on each level has to be decided. Once again, the goal is to optimize the readability of the trace; this is achieved heuristically by minimizing the total length of the arcs. VOV uses a **force-directed** method for dynamical placement of each new node in the trace. If the location determined by the force directed algorithm is already taken, the node is placed in the closest empty location at that level. Although this dynamical placement is suboptimal, it is fast and it can be done on the fly. Upon a user's request, or automatically in periods of inactivity, VOV repeats the one dimensional force-directed placement on one row at a time following it with one pass of **pairwise interchanges** of neighboring nodes to get out of obvious local minima.

The complete placement of a large trace with 1000 nodes and 15 rows takes about a minute on a DECstation 3100. This is a long time, but the data structures used for the trace are not optimized for this placement problem. Five to ten iterations of the algorithm generally yield a reasonably readable trace, that is one where the major flows of the design are easily recognized.

To facilitate the visual inspection of the trace, and to ease the identification of each node, a **short black label** is placed next to each node. The label of a place is obtained by taking the last few components of the place's name, while the label of a transition consists of the first few words in the transition's command line.

4.2 Special topics

4.2.1 Project identification

All VOV clients must establish a connection with the server, which can be running on any host in the network, while that host could be running several servers. Thus, the problem is to assure that the clients connect to the correct server.

Other systems have a similar problem. For example, *NELSYS* [47] also has a server/client architecture, and it identifies the server by means of a *project identifier* of the form *hostname:access-path*, where the component to the left of the colon is the name of the host running the server and the access path is the root directory that contains the entire project.

VOV does not require the design to be contained in the same directory. Instead of the access path, each trace is identified by its *project name*, an arbitrary alphanumeric string, and by the host in which the server is running. The project name is hashed into an 8-bit integer, which is then added a constant to obtain a port number used to initiate the TCP connection with the server.

Each host can support up to 256 projects, provided that their names do not hash to the same integer.

The clients get the project name and the host name by means of two environment variables, `VOV_HOST_NAME` and `VOV_PROJECT_NAME`. If either one is missing, the connection cannot be established and the client continues to run as if VOV did not exist.

4.2.2 Robustness and safety

So much important information is stored in the trace that no precaution to protect the trace can be excessive. VOV keeps two copies of the trace, of which at least one should always be uncorrupted and no more than 30 minutes out-of-date, even in the case of system crash or filling of the disk. In case both copies of the trace become corrupted, VOV has another partial recovery mechanism, based on a compact ASCII dump that can be generated from the trace. This file is in the form of a shell script, and it can be executed directly to rebuild the entire trace, except only for annotations and sets.

The server should be robust to possible misbehavior of its clients. The current implementation of the server can survive any crash of its clients, even during handshaking. It is also not possible for a client to block the server, that is the server will never be in the status of waiting indefinitely for a message from a client.

VOV does not explicitly address the problem of security against malicious intruders. Anyone who knows the project name and the name of the host on which the server is running can connect to the server and force the slaves to do virtually anything. However, all server and slave processes are user processes, without root privileges. Although files of the designers are at risk, the security of the whole computing system is not affected. The data stored in the journal generally allows to track down the intruder. Other ADM's (e.g. [50, 3]) rely on root processes and are therefore increasing the vulnerability of the computer system.

4.3 Software architecture

The main concern in the implementation of VOV was to produce an prototype to test the concepts of a trace-based DMS. Speed and efficiency have also been a concern, because the non-intrusiveness of the system depends in large part on them. However, functionality has had priority over performance, especially because it was not clear, until the system was used, which routines were critical and needed to be optimized.

VOV is written in C++ [48] and it exceeds 27,000 lines of code. It consists of the following seven programs:

vov_server: is the VOV server.

vov_capsule: is used in the encapsulation scripts.

vov_sh: is used as the command based interface of VOV.

vov_assist: is the VOV assistant.

vov_slave: is the VOV slave.

vov_rpc: is the RPC extension to the *Octtools* editor VEM, and provides the graphical user interface.

vov_meter: is the prototypical measurement tool.

These programs operate on the trace through the routines in a library called `trace.a`. A smaller library called `libvov.a` is available for the programs like `vov_meter` that only run as client processes.

Traces are represented by the `VovTrace` class that contains the methods to create, modify and destroy nodes in the trace, as well as methods to alter the connectivity between nodes, to manipulate sets of nodes and so on (Figure 4.3). All routines operate either on a local representation of the trace or on a representation managed remotely by the server. The structure of a typical routine is illustrated in Figure 4.4. Each trace can be either local, client or server. The member functions `isLocal()`, `isClient()`, `isServer()` distinguish the various types. In reference to Figure 4.4, if a client wants to create a new node in the trace it will use the method `VovTrace::create()`, which makes the client send a request to the server and then wait for an answer. When the server gets the request and decides to satisfy it, it executes the same method `VovTrace::create()`, but this time the flow of control is such that the node is actually created. If the trace is local, all the communication protocols are bypassed.

This structure of the routines greatly reduces the amount of code to be maintained, and localizes within the same function the details of the communication protocol between the client and the server.

The information between clients and server is transferred in **packets** of arbitrary length. These packets are sent through UNIX sockets and delivered using the TCP/IP protocol. Packets are processed only when they have been completely delivered. Each packet has a free format, consisting

```

class VovTrace {
    String      projectName;    // This is the name of the project.
    String      hostName;       // Host on which the server is running.
    int         type;           // Server, client, local.
    VovEventQueue workQ;        // Queue for all trace events.
    VovPage*    pageArray;      // Array of pages.
    // ...
    int isClient();
    int isServer( int );
    int isLocal();
public:
    VovStatus open(String& project,String& host,String& mode,int type);
    // ...
    VovStatus create(VovSuperObject&);
    VovStatus find(VovSuperObject&);
    VovStatus modify(VovSuperObject&);
    VovStatus edit(VovSuperObject&,int viewonly=0);
    VovStatus lock(VovObject&,String&,int force,int& Id,String& msg);
    VovStatus unlock(VovObject&,int lockId);
    VovStatus forget(VovSuperObject&);
    VovStatus changeStatus( VovNode&,VovNodeStatus,int flag);
    VovStatus history(VovNode&,int flag,int dir,int lev,String&);
    VovStatus measure(VovPlace& place, VovPlace& measure);

    VovStatus initiate(VovTransition&);
    VovStatus terminate(VovTransition&,VovStatus,int,String&);
    VovStatus dispatchTransition(Client&,VovTransition&);

    VovStatus setOperation(VovNodeSet&,VovNode&,VovSetOperation);
    VovStatus fill(VovNodeSet&,VovNodeSet&,VovSelectRule&,VovNodeSet&);
    VovStatus subselect(VovNodeSet&,VovSelectRule&);

    VovStatus declareInput(VovTransition&,VovPlace&,int);
    VovStatus declareOutput(VovTransition&,VovPlace&,int,int,String&);
    VovStatus declareIOs(VovTransition&,VovPlaceList&);
    VovStatus disconnect(VovNode&,VovNode&);

    VovStatus addAnnotation(VovSuperObject&,VovAnnotation&);
    int      getAnnotations(VovSuperObject&,VovAnnotation*&,int);

    VovStatus checkPlace(VovPlace&,int* exist,Date* timeStamp);
    VovStatus evolve(VovRetrace&,int mode,String&);
    VovStatus evolve(VovRetrace&,VovNode&,int dir,int mode,String&);
    VovStatus stopRetracing(VovTransition&,String&);

    void      doPlacement();
    void      journalLog(const String&);
};

```

Figure 4.3: The VovTrace class contains all the methods to operate on the trace.

```

VovStatus VovTrace::create( VovNode& node )
{
    if ( isClient() ) {
        // Send data to the server.
    } else {
        if ( isServer() ) {
            // Receive data from client.
        }
        // ....
        node.create(); // Actually create the node.
        // ....
        if ( isServer() ) {
            // Send return info to client.
        }
    }
    if ( isClient() ) {
        // Receive return data from server.
    }
    return VOV_OK;
}

```

Figure 4.4: Skeleton of a routine that can be executed either locally by the calling process, or remotely by the server.

of any sequence of numbers and strings, with the only restriction that all packets sent by a client to the server must begin with a number that corresponds to the routine that should be executed by the server.

4.3.1 The hierarchy of classes

A limitation of OCT is that it is not extensible, because it does not allow the definition of new data types. Nevertheless, OCT provides a rich set of primitive objects that, in conjunction with some features of C++, allow the emulation of new types of data used to represent nodes, places and transitions (see Figure 4.5).

The class `VovObject` is a simple C++ wrapper for the `octObject`. A class derived from `VovObject` is `VovProp` that is specialized for the handling of OCT properties. `VovProp`'s are used extensively in the definition of other classes.

The class `VovNode` is derived from `VovObject` and its members are of type `VovProp`. The classes `VovPlace` and `VovTransition` are derived from `VovNode` and are themselves composed of `VovProp`'s. The retrieval of an object from OCT is performed by the member func-

```

class VovObject : struct octObject {
    // ...
};

class VovProp : public VovObject {
    // ...
};

class VovNode : public VovObject {
    VovProp type;
    VovProp level;
    VovProp status;
    // ...
};

class VovPlace : public VovNode {
    VovProp placeType;
    VovProp name;
    VovProp timestamp;
    // ...
};

class VovTransition : public VovNode {
    VovProp cwd;
    VovProp commandLine;
    VovProp affinity;
    // ...
};

```

Figure 4.5: A hierarchy of classes has been defined to describes nodes, places and transitions.

tion `get()`, which retrieves each `VovProp` by name. The retrieval of a place, which must be performed many times during trace operations, requires about 0.5 milliseconds on a DECstation 3100. In order to speedup the operation of the server, future implementations should exploit the fact that for each particular task not all the fields in a objects are needed, and is should replace the indiscriminate retrieval of all `VovProps` with a selective retrieval of only the necessary fields on demand.

Virtual functions are used rarely and all objects are explicitly given a type. Virtual functions use information that is managed by C++ and not accessible by the programmer; this information would be lost when objects are transferred between clients and server through sockets.

4.3.2 User interface

The designers have access to the trace through three interfaces differing in weight and in power: a command interface, a menu-driven interface, and a graphical interface. The primary objective of these interfaces is to minimize the number of user actions such as key-strokes, mouse pointing and clicking, required to perform the most common tasks, and to minimize the mixing of keyboard and mouse actions within a single task.

The most used interface is a lightweight command based interface that requires only a simple alphanumeric terminal. Command based interfaces are useful because they can be programmed and extended. This interface allows control and monitoring of retracing, editing of the trace, querying about the history of places in the trace. The program `vov_sh` and its many options (Figure 4.6) are the key elements of this interface.

The second interface is based upon pop-up dialogs and requires a terminal supporting X windows. This interface is easier to use and more powerful, but it cannot be programmed because it requires pointing and clicking. The control panel for this interface is shown in Figure 4.7. The user can edit and create sets, control retracing and the activity of the slaves, browse the trace and inspect the event queue. All dialogs have an "Help" button that activates a subordinate dialog containing a textual description of the dialog and its functions.

Since the trace is itself stored as an OCT facet, it was easy to develop a graphical user interface capitalizing on the OCT editor VEM and its Remote Procedure Call (RPC) mechanism. VEM is a multi-window graphical editor, that can be used to browse the trace and to follow the flow of the tools. The RPC mechanism allows VEM to be extended with some commands specific to VOV, such as the command `view` that can be issued whenever the mouse pointer is over a node to pop-up a dialog describing the attributes of the node. Other RPC commands allow the editing of the status of the nodes and of the connectivity of the trace. Nodes can be selected and connected to other nodes, or deleted from the trace.

The performance of OCT does not affect the performance of VOV, which is dominated by the interaction with the file system. The only unsatisfactory performance has been observed in the RPC interface, which slows down some data base intensive operations, such as the placement of the trace, by a factor of 10 or more.


```

usage: vov_sh [--E on_error] [-ADEF] [-H params] [-LK] [-N string] [-O string]
      [-RSVd] [-p project] [-t name] [-u name] [-w] [-f place]
      [-e place] [-r place] [-k place] [-h name] [-mniI]
--E:      cause fatal errors to core dump (on_error = "core") or exit
          (on_error = "exit")
-A:      Advanced user flag
-D:      Dump trace in ASCII format (used for emergency save)
-E:      Show all events during retracing
-F:      Set speed of retracing to FAST (default SLOW)
-H:      History : ameters. Examples: -T30, +N2, -TP1000.
-L:      Local trace (expert users only).
-K:      Kill server
-N:      New string for mvlib function
-O:      Old string for mvlib function
-R:      Retrace all
-S:      Stop all retracing
-V:      Print version number
-d:      Debug communication with server
-p:      Specify a different project name than VOV_PROJECT_NAME.
-t:      Toggle status of place
-u:      Is the place used in the design?
-w:      Write trace onto disk (also force check of all places)
-f:      Forget place (Use with caution!!)
-e:      Edit place
-r:      Retrace TO specified place (retrace the place)
-k:      Retrace FROM specified place (kick the place)
-h:      Print history of place
-m:      Monitor slaves
-n:      Notification: get all unreported events
-i:      Interactive
-I:      Get info about design

```

Figure 4.6: The usage message generated by the vov_sh shows the options that give access to many of VOV's services.

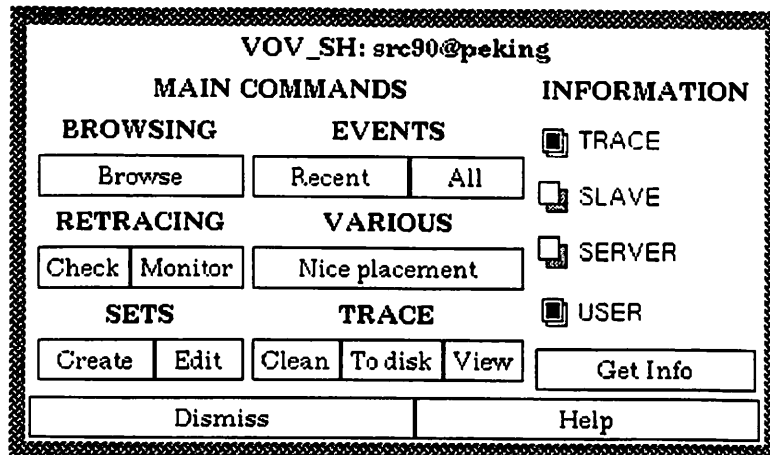


Figure 4.7: The control panel for the menu driven interface to VOV puts the most common operations only a few mouse clicks away.

4.4 Performance

4.4.1 Server latency

The server can become a performance bottleneck of the system, because all accesses to the design trace must go through it. For a client, the **server latency** is the time required between the submission of a request for service and the arrival of the response from the server. The latency should be minimized.

The most common server/client interaction is the declaration of inputs and outputs of a transition, which may become critical when many transitions are executing concurrently. For example, in the recompilation of VOV, there can be up to about twenty files being compiled in parallel, with each compilation declaring thirty or more inputs.

For each declaration the server must perform:

- a search by name in the set of all places to determine whether the place is already in the trace;
- some checks for input, output and lock conflict (see Section 3.9.4);
- a cycle detection check (also in Section 3.9.4).

The search by name uses hash tables and requires a constant time, and the detection of conflicts is also fast. The critical step is cycle detection.

Given a trace $\mathcal{T} = (T, P, E)$ and the declaration of a new arc $(t, p), t \in T, p \in P$, we want to know if the new trace $\mathcal{T}' = (T, P, E'), E' = E \cup \{(t, p)\}$ contains cycles. A worst case

Cycle detection work		
Trace name	Nodes	Max nodes visited
octtest	369	17
vovmips	616	30

Table 4.1: Experimental observation of the maximum work required by the cycle detection routine.

analysis shows that all the arcs in the trace may have to be traversed, so that the amount of work necessary for cycle detection is $O(|E|)$, with $|E| \leq |P|^2/2$. But this is pessimistic, because only the arcs reachable from t or p need to be visited. In practice traces tend to be wide and shallow, so that the number of nodes reachable from any node is a small fraction of all the nodes, as shown in Table 4.1. In most cases the cycle detection can be resolved immediately because p has no inputs or no outputs, and cannot therefore belong to a cycle.

We have measured the throughput of the server for common operations such as input and output declarations. Those measurements on a DECstation 3100 show that each declaration requires between 0.01 and 0.10 seconds (elapsed time), which means that the server can process between 10 and 100 declarations per second. The most complex transitions observed to date (e.g. compilations or floorplanning) declare close to 40 inputs and outputs, in which case the overhead due to the server's operation is less than 4 seconds, usually a negligible fraction of the transition duration. Most transitions declare between 3 and 6 inputs and outputs.

The actual overhead seen by each tool depends on the latency due to the TCP/IP protocol and the number of clients concurrently competing for services. In order to limit the number of round trips between a client to the server and back, an entire array of input and output places can be declared at once.

4.4.2 Capsule overhead

In the current implementation, a capsule is a shell script consisting of two parts:

1. The first part computes the inputs and outputs of the transition, using information extracted from the command line arguments, and possibly even parsing some of the input data. The same operations will be performed by the tool itself.
2. The second part is a call to the program `vov_capsule`, which takes care of all communication with the server and then forks a new process to execute the actual tool with the same

Capsule overhead in seconds						
Tool	Elapsed time				Overhead	
	Tool		Capsule			
	min	max	min	max	min	max
bdsyn	1	4	9	16	5	17
misII	14	20	23	32	3	18
bdnet	3	8	17	19	9	16
padplace	1	3	5	16	2	15
wolfe	87	109	103	105	0	18

Table 4.2: Capsule overhead for some brief transitions.

arguments used to call the capsule. The overhead introduced by `vov_capsule` is related to the server latency.

Table 4.2 shows the results of an experiment to measure the capsule overhead for some transitions of short duration. Each tool has been run three times in rapid succession, followed by three runs of the encapsulated tool, always measuring the elapsed time. A DECstation 3100 running only the server and the transition was used for these measurements. The operating system is responsible for variations of several seconds in the elapsed time of repeated experiments. Other measurements have determined that the largest contribution to the overhead is due to the interpretation of the encapsulation script. The capsule overhead is in the range of 2 to 20 seconds for most of the *Octtools*. Users of VOV considered this overhead to be too large. The reduction of this overhead is an important objective in the further refinements of the system, in a continuing effort to make the system as non-intrusive as possible. The best solution, of course, will be recompilation of the tools with the VOV library.

4.4.3 Trace size

Table 4.3 shows the size of the disk representation of the trace for a few designs. The important column is the third, showing the number of kbytes per node. Each node requires between 610 bytes and 910 bytes of memory on disk. The larger number is from the trace of the compilation of VOV, which contains many transitions with long command lines (more than 80 characters), and several transitions with extremely long commands (more than 400 characters). The in-core requirements for each node are difficult to measure, but they are estimated to be 3 to 4 times those for the persistent disk representation.

Disk usage for the trace			
Nodes (places+transitions)	Size (kbytes)	kbytes/node	Comments
0 (0+0)	2	-	Empty trace (overhead)
3 (2+1)	4	0.73	Smallest possible trace
23 (18+5)	16	0.61	4 slides
35 (27+8)	26	0.69	One cell design
201 (157+44)	122	0.60	40 slides
254 (208+46)	160	0.62	Compilation
576 (453+123)	532	0.92	VOV compilation
674 (484+190)	616	0.91	VOV compilation
1294 (1012+282)	792	0.61	DSP chip

Table 4.3: Memory usage for disk representation of trace. The overhead due to the requirements of the empty trace is subtracted from the trace size before computing the memory requirements per node.

4.4.4 Small designs

Designers do not ask for any assistance while performing simple activities such as running three tools in sequence, because they believe that they can easily handle it. This became clear only after talking to some students in a VLSI class who preferred not to use VOV while doing their homeworks, for the simple reason that the homework itself was straightforward.

Small designs are those in which the overhead introduced by the capsules and by the server/client communication is most prominent. Most designers form their first opinion about a DMS using simple tests, which makes the performance of the system on small designs critical for the acceptance of the system.

The transition between a small hand-managed design and a more complex one requiring automatic assistance is a good test of the non-intrusiveness of a DMS. In VOV this transition requires only the activation of the tracing mechanism: all tools remain exactly the same, the designers do not have to switch to new tools or to new names for the same tools.

4.4.5 Large designs

To date, the largest designs managed by VOV have traces consisting of less than 2000 nodes, and for these designs there is no appreciable degradation in the performance of the server. However, we can expect designs many times more complex, with 10 or 20 thousand nodes in the

trace. This raises the questions of whether VOV will scale gracefully, and if not of what can be done to handle large designs.

It is definitely possible to improve the efficiency of the current implementation, for example by streamlining the communication protocol and optimizing the retrieval of nodes from the database. But there are other possibilities: partitioning the trace into pages, and having multiple servers.

In the current implementation the entire trace is held in one OCT facet, called `page . 0`, which is always loaded in the virtual memory of the server. Since only a few nodes in the trace show activity at any one time, it would be advantageous to limit the memory size of the server by keeping in `page . 0` only the active nodes, while all other nodes are pushed onto other pages that are normally stored on the disk. The main difficulty in this approach lies in the need to move rapidly an inactive node into `page . 0` as soon as it becomes active, which can happen suddenly, without warning.

Multiple servers can also cooperate for the same design. For example, it is possible to adopt the recommendation by Katz et al. in [15] that the *design process* consists of several *design activities*, each with its own history, and have one server for each activity. The rule for cooperation is that the set of design transitions must be partitioned among the servers to avoid duplication.

Communication among servers is achieved through the data. For example, consider a place that is a primary input for one server *S2* and the output of a transition for another server *S1*. When the transition is executed, the place changes its timestamp. The server *S2* detects the change in the place, and it can therefore invalidate and then retrace all the nodes that depend upon the place in its own trace. No server has a complete representation of the dependencies among design data, and data consistency is only guaranteed when each server believes that its own data are consistent.